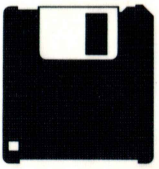
Markt&Technik

Dr. Edgar Huckert/Frank Kremser

AMIGA C in Beispielen

Grundlagen ★ Mit umfangreicher Programmsammlung für den C-Programmierer ★ Aufruf der Systembibliotheken unter C



Auf 3½"-Diskette enthalten:
Alle C-Beispielprogramme aus dem Buch im Source-Code und ablauffähig kompiliert.

Amiga: C in Beispielen

Dr. Edgar Huckert/Frank Kremser

AMIGA

C in Beispielen

Grundlagen ★ Mit umfangreicher Programmsammlung
für den C-Programmierer
★ Aufruf der Systembibliotheken unter C

Markt&Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Huckert, Edgar
Amiga C in Beispielen : Grundlagen, mit umfangreicher Programmsammlung für d. C-Programmierer, Aufruf d. Systembibliotheken unter C / Edgar Huckert. –
Haar bei München : Markt-u.-Technik-Verl., 1987.
ISBN 3-89090-539-0

Die Informationen im vorliegenden Produkt werden ohne Rücksicht auf einen eventuellen Patentschutz veröffentlicht.
Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt.
Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter Sorgfalt vorgegangen.
Trotzdem können Fehler nicht vollständig ausgeschlossen werden.
Verlag, Herausgeber und Autoren können für fehlerhafte Angaben und deren Folgen weder eine juristische Verantwortung noch irgendeine Haftung übernehmen.
Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der Speicherung in elektronischen Medien.
Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten ist nicht zulässig.

»Commodore-Amiga« ist eine Produktbezeichnung der Commodore Büromaschinen GmbH, Frankfurt, die ebenso wie der Name »Commodore« Schutzrechte genießt.
Der Gebrauch bzw. die Verwendung bedarf der Erlaubnis der Schutzrechtsinhaberin
Amiga ist eine Produktbezeichnung der Commodore-Amiga Inc., USA
Microsoft-C ist ein eingetragenes Warenzeichen der Microsoft Corporation, USA
Lattice-C ist ein eingetragenes Warenzeichen der Lattice Corporation, USA

15 14 13 12 11 10 9 8 7 6 5 4 3 2
90 89 88 87

ISBN 3-89090-539-0

© 1987 by Markt&Technik Verlag Aktiengesellschaft,
Hans-Pinsel-Straße 2, D-8013 Haar bei München/West-Germany
Alle Rechte vorbehalten
Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Huber, Dießen
Printed in Germany

Inhaltsverzeichnis

Vorwort 1 9
Vorwort 2 11

1 Der Amiga und C 15

2 Warum C? 25

3 Erste C-Programme 29
3.1 Programmabdruck 29
3.2 Kommentare 30
3.3 Zusammenfassung 34
3.4 Kompilieren, Linken, Starten 35
3.5 Etwas C-mäßiger formuliert 36

4 Daten und Datentypen 41
4.1 Daten und Befehle 41
4.2 Konstanten und Variablen 42
4.3 Datentypen 43
4.3.1 Wozu Datentypen? 43
4.3.2 Hierarchien von Datentypen 45
4.3.3 Eigenschaften der Basistypen 47
4.3.4 Konvertierungsregeln 48
4.4 Schreibregeln 51
4.4.1 Schreibregeln für Konstanten 51
4.4.2 Schreibregeln für Variablen und andere C-Objekte 54
4.5 typedef 56
4.6 Gültigkeitsbereiche und Speicherklassen 56

5	**Operatoren**	59
5.1	Ausdrücke und Operatoren	59
5.2	Die einzelnen Operatoren	60
5.2.1	Der Funktionsaufruf ()	60
5.2.2	Auswahl eines Arrayelements []	61
5.2.3	Der Punktoperator: Auswahl eines Strukturelements	61
5.2.4	Der Pfeiloperator: Auswahl eines Strukturelements	62
5.2.5	Typumwandlung: Der Cast-Operator	62
5.2.6	Der sizeof-Operator	63
5.2.7	Der Adreßoperator &	64
5.2.8	Der Sternoperator *	64
5.2.9	Der Vorzeichenoperator -	65
5.2.10	Die logische Negation !	65
5.2.11	Der Komplementoperator ~	65
5.2.12	Der Additionsoperator +	66
5.2.13	Der Subtraktionsoperator -	66
5.2.14	Der Multiplikationsoperator *	66
5.2.15	Der Divisionsoperator /	67
5.2.16	Der Restoperator %	67
5.2.17	Der Shiftoperator >>	67
5.2.18	Der Shiftoperator <<	68
5.2.19	Der logische Oder-Operator ||	68
5.2.20	Der logische Und-Operator &&	68
5.2.21	Der bitweise Oder-Operator |	69
5.2.22	Der bitweise Und-Operator &	69
5.2.23	Der bitweise Exor-Operator ^	69
5.2.24	Der Gleichheitsoperator ==	69
5.2.25	Der Ungleichheitsoperator !=	70
5.2.26	Die Zuweisungsoperatoren	70
5.2.27	Der Fragezeichenoperator ?	71
5.2.28	Der Kommaoperator ,	72
5.3	Vorrangregeln für Operatoren	72
6	**Befehle**	75
6.1	Befehlstypen	75
6.2	Deklarationen	76
6.3	Zuweisungen	78
6.4	Prozeduraufrufe	79
6.5	Bedingungen	79
6.6	Schleifen	83
6.7	Der Sprungbefehl	87
6.8	Zusammenfassung	89

7	**Eingabe und Ausgabe**	91
7.1	Die Rolle der C-Bibliothek	91
7.2	UNIX-Philosophie der Ein-/Ausgabe	92
7.3	Familien von Ein-/Ausgabefunktionen	92
7.4	Umlenkung der Ein-/Ausgabe	99
7.5	Prozeßkommunikation	101
8	**Arrays und Pointer**	105
8.1	Pointer	105
8.2	Eindimensionale Arrays	110
8.3	Die Parallelität von Arrays und Pointern	112
8.4	Mehrdimensionale Arrays	116
9	**Strukturen und Verbunde**	121
9.1	Begriffbestimmungen	121
9.2	Deklaration und Initialisierung	123
9.3	Zulässige Operationen	125
9.4	Selbstbezügliche Strukturen	127
9.5	Ein Testprogramm für Strukturen	130
10	**Prozeduren**	133
10.1	Grundlegendes zu Prozeduren	133
10.2	Argumentübergabe	138
10.3	Schnittstellen zum Betriebssystem	140
10.4	Fortgeschrittene Anwendungen von Prozeduren	141
10.4.1	Beliebig viele Argumente	141
10.4.2	Rekursive Prozeduren	145
10.4.3	Funktionen mit besonderen Rückgabewerten	148
10.4.4	Pointer auf Funktionen	150
10.5	Zusammenfassung	150
11	**Der C-Präprozessor**	151
11.1	Funktionen des Präprozessors	151
11.2	Definieren von Konstanten und Makros	152
11.3	Einkopieren von Quellcode	156
11.4	Bedingtes Kompilieren	158
11.5	Zusammenfassung	161
12	**Die C-Bibliothek**	163
12.1	Einführung in die C-Bibliothek	163
12.2	Die wichtigsten Funktionen der C-Bibliothek	164
12.3	Ein Testprogramm für Bibliotheksfunktionen	184

13	**Programmierstil in C-Programmen**	189
13.1	Warum Programmierregeln?	189
13.2	Elementare Strukturregeln	190
13.3	Zerlegung in Prozeduren	190
13.4	Wahl von globalen und lokalen Variablen	191
13.5	Namensvergabe	192
13.6	Weitere Schreibregeln	193
14	**Systemprogrammierung**	199
14.1	Begriffsbestimmung	199
14.2	Pflege von Systemtabellen	200
15	**Programmoptimierung in C**	203
15.1	Möglichkeiten der Optimierung	203
15.2	Verwendung von Registervariablen	204
15.3	Großer und kleiner Code	204
16	**Die ANSI Norm für C**	207
17	**Eine kleine Sammlung von Beispielsprogrammen**	209
17.1	Konvertierroutinen	209
17.2	C im kommerziellen Einsatz: DIN-gerechte Sortierung	215
18	**Kleines Glossar**	227
Stichwortverzeichnis		233
Hinweise auf weitere Markt&Technik-Produkte		238

Vorwort 1

Das vorliegende Buch soll eine praxisgerechte Einführung in C bieten. C hat sich in den letzten Jahren als die wichtigste universelle höhere Programmiersprache herausgestellt. Derzeit werden viele kommerzielle Standardprogramme in C umgeschrieben. Da C nicht in Dialekte zerfallen ist, bietet diese Programmiersprache fast eine Garantie für leichte Transportierbarkeit – und das ist angesichts der ständigen Generationswechsel im Hardwarebereich lebenswichtig für Programme. Nicht zuletzt die zunehmende Akzeptanz von UNIX unterstützt die weite Verbreitung von C, das ja sozusagen die Hausprogrammiersprache von UNIX ist. Im PC-Bereich wird MS-DOS immer UNIX-ähnlicher; das Betriebssystem des Amiga und das TOS des ATARI ST sind ohnehin weitgehend in C geschrieben, was auch für das Betriebssystem des AMIGA gilt!

Das Buch ist nicht für Programmieranfänger geschrieben. Zumindest elementare Programmierkenntnisse sollten beim Leser vorhanden sein. Um auch Umsteigern aus anderen Programmiersprachen das Lesen zu erleichtern, wurde auf einen rein systematischen Aufbau verzichtet. Statt dessen habe ich das Modell amerikanischer Lehrbücher gewählt, in denen auf ein Kapitel 'getting started' die systematischen Kapitel folgen. Wer also schnell zu einem laufenden C-Programm kommen will, sollte sich das Anfangskapitel 'Erste C-Programme' ansehen und dann das Buch eher im Sinne eines Handbuchs benutzen. Lesern mit weniger Erfahrung rate ich, das Buch von vorne nach hinten durchzuarbeiten.

C kann sicher nicht durch reines Buchstudium gelernt werden, da es viel liberaler und damit fehleranfälliger als andere Programmiersprachen ist. Deshalb rate ich dringend, die wichtigeren Beispiele am eigenen Computer durchzurechnen. Ich habe mich bemüht, die Beispiele maschinenunabhängig auszuwählen. Fragen der Portabilität spielen eine große Rolle. (Anmerkung: Alle aufgeführten Listings sind für den AMIGA geändert.)

Ungewöhnlich für ein Lehrbuch ist sicher der Nachdruck, der auf die Stilfragen der Programmierung gelegt wird. Da C eher zur knappen Ausdrucksweise tendiert und bisher eher im professionellen, zur lakonischen Formulierung tendierenden Bereich angesiedelt war, stehen C-Programme im Ruf, schlecht lesbar zu sein. Ich hoffe zeigen zu können, daß man dem leicht abhelfen kann.

Im letzten Kapitel habe ich einige Beispielprogramme gesammelt und kommentiert. Diese Programme sollen konsistente Beispiele für die Programmierung in C bieten; ich garantiere nicht, daß die Programme alle Bedürfnisse der Praxis für das jeweilige Einsatzgebiet abdecken. In den vorhergehenden Kapiteln sind immer wieder längere, vollständige Programme abgedruckt, die durchaus in der Praxis verwertbar sind.

Das Buch stützt sich mit einer Ausnahme (Eindeutigkeit der Namen von Strukturelementen) auf den Kernighan & Ritchie (K&R)-Standard. Die kommende ANSI-Norm für C ist noch nicht verabschiedet und bietet meines Erachtens keine revolutionären Erweiterungen, so daß ich die Beschränkung auf Kernighan & Ritchie rechtfertigen kann.

Für Kritik und Anregungen bin ich dankbar.

Dr. Edgar Huckert
Ringofenstr. 23
8758 Goldbach

Vorwort 2

Der Amiga verfügt über großartige Möglichkeiten, die bisher nur in wesentlich kostspieligeren Geräten vorzufinden waren. Dazu gehören beispielsweise die vorzügliche Grafik, hervorragende Soundfähigkeiten und Multitasking.

Zudem hält der Amiga zwei verschiedene Benutzeroberflächen bereit. Zum einen ist dies die sogenannte Workbench, auf der Dateien mittels Icons gehandhabt werden. Auch sämtliche anderen Funktionen werden grafisch gelöst, so daß für den Computerneuling der Einstieg erheblich vereinfacht wird. Die zweite Benutzeroberfläche orientiert sich an den herkömmlichen Methoden und ist tastaturorientiert. Gemeinsam haben beide Methoden, daß sie die Multitaskingfähigkeit des Amiga vorbildlich unterstützen.

Der Amiga stellt aber nicht nur für den Benutzer ein hervorragendes Gerät dar. Auch Programmierer haben ihre helle Freude daran. Man hat alle Mühe darauf verwandt, den Zugriff auf das komplexe System zu erleichtern. Wo immer möglich, werden die von der Implementierung abhängigen Komponenten durch Treiberroutinen angesprochen. Das bedeutet, daß der Programmierer in die Maschinenabhängigkeit soweit hinabsteigen kann, wie es für seinen Zweck dienlich ist. Um das Ansprechen der Treibersoftware zu erleichtern, werden Standard-Parameter verwendet.

Mit der Fähigkeit, mehrere Aufgaben zugleich bearbeiten zu können, ragt der Amiga über die meisten kleineren Computersysteme hinaus. Beim Übergang vom Großcomputer zum Personalcomputer vermissen viele erfahrene Programmierer die Möglichkeit, mehrere Aufgaben gleichzeitig ablaufen zu lassen. Wenn diese Möglichkeit (wie beim Amiga) dennoch gegeben ist, so liegt der Vorteil für die Produktivität klar auf der Hand: Man kann z.B. eine neue Datei editieren, während im Hintergrund noch ein Compiler läuft.

Die Hardware des Amiga ist einzigartig, denn beim Entwurf des Amiga wollten die Entwickler spezielle Hardware einsetzen, wo immer dies vernünftig erschien. Dies ist zweifellos gelungen. Gemeinsamkeiten bestehen zwar mit allen kleineren Computern, insofern eine Zentraleinheit (CPU) alle Aktivitäten der Maschine kontrolliert, die Ein- und Ausgabe, die Speicherverwaltung u.s.w. Die Zentraleinheit des Amiga besteht allerdings aus dem MC68000 von Motorola, der äußerst leistungfähig ist. Aber dies ist nicht alles, denn die CPU wird von einer Reihe von Koprozessoren, auch Custom-Chips genannt, unterstützt. Diese Chips nehmen der CPU eine große Anzahl zeitraubender Arbeiten ab. So kontrolliert der Copper das grafische System, während der Blitter auf schnelle Datenübertragung spezialisiert ist. Aber noch weitere Aufgaben werden der CPU abgenommen, beispielsweise auch die Klangerzeugung.

Die Anordnung der unterstützenden Prozessoren und Chips trägt zu der wirksamen Arbeitsweise bei. Der Amiga ist ein busorientiertes System. Beim Amiga 1000 befindet sich dieser Bus an der rechten, beim 500 an der linken Seite und beim 2000 ist er im Gerät an einem Slot zu finden. Alle Spezialchips und Koprozessoren stehen mit dem Bus in Verbindung. Die meisten haben überdies einen prozessorunabhängigen Zugriff auf den Hauptspeicher. Man nennt dies DMA (engl. Direct Memory Access). Weiterhin wird jeder zweite Systemtakt den Spezialchips zur Verfügung gestellt. Die CPU läuft aber trotzdem mit voller Geschwindigkeit weiter, wobei die Geschwindigkeitsunterschiede zwischen ihr und dem Speicher genutzt werden. Es ist allerdings erwähnenswert, daß der Bus bei einigen Operationen des Blitters selbst der CPU vorenthalten bleibt.

Die Regelung des direkten Speicherzugriffs erfolgt durch Gerätetreiber. So kann z.B. die Floppy Daten direkt zwischen Speicher und Diskette übertragen, ohne diese über einen Datenpuffer zu leiten. Das Ein-/Ausgabesystem arbeitet nach dem gleichen Prinzip. Einen solchen Aufbau konnte man früher nur bei großen Computern vorfinden. Es gibt allerdings auch Einschränkungen in diesem System: Der Amiga kann zwar einen Speicher von mehr als 8 Mbyte verwalten. Der DMA-Zugriff der spezialisierten Hardware ist allerdings auf die unteren 512 Kbyte beschränkt.

Der Amiga verfügt aber nicht nur über eine hervorragende Hardware, sondern auch über viel Software zum Ansprechen dieser Hardware, was durch die Kickstartgröße von 256 KByte eindrucksvoll dokumentiert wird. Diese Software hat folgende Besonderheiten: Es werden für Programme keine absolut festliegenden Adressen im Speicher oder in der Hardware benutzt. Da es sich beim Amiga um ein Multitasking-System handelt, kann man nie davon ausgehen, daß sich bestimmte Software, nicht einmal die Systemsoftware, ab einer bestimmten absoluten Adresse befindet. Sie wird vielmehr relativ zu einem Zeiger abgelegt. Der Zugang zu Systemroutinen erfolgt fast ausschließlich über Pfade in der Software.

Der Amiga besteht aus drei verschiedenen Betriebssystemen, die jedoch eng zusammenarbeiten. Diese sind:

- der sogenannte Kernel (Kern), ein ausführendes Betriebssystem, auf der untersten Ebene mit einem Minimum an Funktionen. Zu ihnen gehören die Zuweisung von Speicherplatz, der Nachrichtenaustausch zwischen ablaufenden Prozessen und Geräten, die Basis des Multitasking und die primären Aufgaben der Ein- und Ausgabe.
- AmigaDOS, ein traditionelles Betriebssystem, das die Dateiverwaltung und das Multitasking auf hoher Ebene übernimmt.
- Intuition, die vorrangig eingesetzte Benutzeroberfäche mit Ikonen und Fenstern.

Der ausführende Kernel unterstützt sowohl AmigaDOS wie auch Intuition. AmigaDOS ist das Betriebssystem der zweiten Ebene. Es stellt dem Anwender Dienste insbesondere für Dateien zur Verfügung. Intuition greift auf diese Dateifunktionen zurück und ist insofern abhängig vom AmigaDOS. Die Abhängigkeit ist jedoch nicht streng hierachisch, denn Intuition greift auch direkt auf den ausführenden Kernel zurück. In der Praxis verursacht dieses Übereinandergreifen jedoch keinerlei Probleme für den Anwender oder Programmierer.

Als Programmierer kann man sich ungezwungen durch alle drei Ebenen der Betriebssysteme bewegen und ihre Dienste in Anspruch nehmen, da sie sich nicht gegenseitig ausschließen. In dieser Programmierumgebung lassen sich leistungsfähige Programme schnell und leicht entwickeln. Wenn eine Systemroutine nicht die gewünschten Dienste erbringen sollte, so kann man sie ohne weiteres durch Module aus tieferen Ebenen ersetzen, ohne das gesamte Programm neu aus solchen Modulen zusammensetzen zu müssen. Assemblerprogrammierung wird nur in wenigen Fällen notwendig werden, da C sehr eng mit dem Betriebssystemen zusammenarbeitet. Viele Funktionen werden in sogenannten Libraries (Bibliotheken) verwaltet. Die Programmiersprache C, die für anspruchsvolle Programme auf dem Amiga Verwendung findet, unterstützt diese Bibliotheken in vorbildlicher Weise. Aber die Sprache C bietet darüber hinaus natürlich ebenfalls die Grundfunktionen, die auch in anderen C-Versionen vorhanden sind. Diese Grundfunktionen stellen einen Bestandteil dieses Buches dar. Zudem wird der Leser in die Programmiertechnik der Sprache C eingeführt.

Die Vorteile dieser Sprache liegen klar auf der Hand: Da ist zum einen die leichte Portabilität anzuführen. Das bedeutet, daß C-Programme, die auf einem anderen Rechner erstellt wurden, verhältnismäßig leicht übertragen werden können. Ein weiterer Vorteil ist der schnelle Maschinencode, den der C-Compiler liefert. Das heißt, der Programmierer kann alle Vorteile ausnutzen, die die strukturierte Programmierung bietet. Trotzdem erhält er anschließend ein überaus schnelles Programm.

Um genauer auf das Amiga-System eingehen zu können, reicht leider der Platz in diesem Buch nicht aus. Aus diesem Grund möchte ich Sie auf die Bücher 'Das Amiga-Programmierhandbuch' und 'Das Amiga-Hardwarebuch' verweisen, beide im Markt&Technik-Verlag erschienen, falls Sie weitere Informationen benötigen, oder aber ein wenig tiefer in die Materie einsteigen wollen.

Dieses Buch stellt eine Übertragung des Buches 'C in Beispielen' dar, das überwiegend auf den IBM-PC und Kompatible ausgerichtet war. Es wurde allerdings in der Weise umgestellt, daß die compilerspezifischen Möglichkeiten von Lattice- und Aztec-C berücksichtigt wurden.

Frank Kremser

1 Der Amiga und C

Der Commodore Amiga ist ein vielseitiger Computer. Somit fällt es vielen sehr schwer, diesen Rechner in die richtige Kategorie einzuordnen. Seine Anwendungsgebiete reichen vom professionellen Einsatz als Personal Computer im Business-Bereich bis zum idealen Rechner für Programmierer.

Die alten, nicht bedienungsfreundlichen Programme werden nun bald ein Teil der Geschichte sein. Umständliche Tastaturbedienung von Programmen, Schwarzweiß-Grafiken oder solche mit vier Farben, Grafiken mit niedriger Auflösung, langsame Geschwindigkeiten und ein öder Ablauf von Programmen gehören nun, dank des Amiga, der Vergangenheit an. Schnelle Grafiken und hohe Rechengeschwindigkeiten, Mausbedienung sowie Multitasking beherrschen nun das Geschehen auf dem Software-Markt, nach dem sich jeder Hobby- oder Profiprogrammierer richten muß, wenn seine Programme großen Zuspruch bei den Software-Anwendern finden sollen. Ich persönlich sehe im Amiga eine faszinierende Maschine für Programmierer, die viele Reize und Geheimnisse enthält.

Mit C auf dem Amiga zu programmieren, zeigt am deutlichsten die Vorteile, die diese Sprache bietet. So können alle Funktionen, die die Hardware bietet, über C angesprochen werden. Ein Großteil der Leistungsfähigkeit resultiert aus dem stark modularisierten Gesamtkonzept dieses Rechners.

Wie schon zuvor erwähnt, können verschiedene systemspezifische Routinen in sogenannten Include-Files abgelegt werden, die dann mit '#include' in ein eigenes Programm eingebunden werden können. Dies stellt beim Amiga aber nur eine Möglichkeit dar, spezifische Eigenschaften für ein eigenes Programm nutzbar zu machen. Zwei weitere überaus wichtige Möglichkeiten, die übrigens Amiga-spezifisch sind, stellen die 'Libraries' und 'Devices' dar. Üblicherweise nehmen sie in der Programmierung auf dem Amiga einen sehr hohen Stellenwert ein. Dieses System hat allerdings auch seine Nachteile, die nicht verschwiegen werden sollen. Denn Programme,

die diese Libraries und/oder Devices benutzen, sind nicht mehr so leicht auf andere Rechner zu übertragen. Und gerade das ist ja die eigentliche Stärke von C.

Auf dem Amiga hat sich C als Programmiersprache für anspruchsvolle Programme durchgesetzt. Dies liegt vornehmlich daran, daß C-Programme enorm schnell, dabei aber dennoch recht einfach zu erstellen sind.

Für den Amiga stehen derzeit zwei verschiedene C-Compiler zur Verfügung. Dies sind der Manx-Aztec- und der Lattice-C-Compiler. Der Unterschied liegt im Preis und der Bedienungsfreundlichkeit sowie in der Schnelligkeit und Kompaktheit des erzeugten Codes. Beide lassen sich leider nur vom CLI, dem 'Command Line Interface' des Amiga, bedienen, was anfangs sehr mühsam ist. Wir möchten Ihnen an dieser Stelle keinen Compiler anraten, da dies schon in großer Zahl in den Fachzeitschriften geschehen ist.

Wir möchten jedoch jedem Programmierer dazu raten, eine Festplatte zu verwenden, sei es eine Amiga- oder eine PC-Harddisk im SideCar, bzw. Amiga 2000, da das Kompilieren mit Disketten äußerst zeitraubend ist. Außerdem besitzen Disketten eine für C nicht gerade üppige Speicherkapazität, so daß es sehr schnell zu einem Speichermangel kommen kann.

Zur Vereinfachung des Compiliervorganges habe ich ein Batch-File für den Lattice-C-Compiler geschrieben, das alle Compilier- und Linkphasen selbstständig durchführt (der Aztec-Compiler stand mir leider nicht zur Verfügung, aber im Benutzerhandbuch düften die einzelnen Schritte genau erklärt sein):

```
.key prg
stack 20000
if not exists <prg>.c
   echo "File ist nicht vorhanden"
   skip end
   endif
echo "-- Compilieren --"
lc1 -i:include/ <prg>.c
if not exists <prg>.q
  echo "Compilier-Fehler"
  quit 20
  endif
lc2 <prg>
alink : lib/lstartup.obj+<prg>.o library :
lib/lc.lib+:lib/Amiga.lib to <prg>
delete <prg>.o
echo "-- Compilier und Linkvorgang ist zu Ende --"
lab end
```

Wenn Sie Besitzer der Version 3.40 sind, so muß die Zeile, die mit 'alink' beginnt, folgendermaßen geändert werden:

```
blink : lib/c.o+<prg>.o library : lib/lc.lib+:lib/Amiga.lib to
<prg>
```

Diese Datei muß mit dem Editor 'ed' eingegeben und gespeichert werden. Dazu gehen Sie von der Workbench aus in das CLI. Dort erscheint '1>'. Nun kommen Sie mit 'ed comp' in den Editor und können das Batch-File eingeben. Wenn Sie fertig sind, speichern Sie es ab, indem Sie 'ESC' und anschließend 'x' drücken. Das Batch-File steht nun unter dem Namen 'comp' auf Ihrer Diskette, bzw. Harddisk. Wenn Sie später ein selbstgeschriebenes C-Programm kompilieren wollen, starten Sie das Batch-File mit dem CLI-Befehl 'EXECUTE comp' und den Programmnamen des C-Programms ohne das '.c'-Kürzel. Was bewirkt nun dieses Batch-File. Zu Beginn wird der Programmname der Variablen 'prg' übergeben und anschließend der Stack auf eine Größe von 20000 Bytes gesetzt. Dies ist nötig, da der Compiler eine Vielzahl von Daten zwischenspeichern muß. Anschließend überprüft es, ob das gewünschte Programm zum Kompilieren überhaupt existiert. Ist das Programm nicht vorhanden, steigt das Batch-File aus und druckt die Fehlermeldung 'File ist nicht vorhanden'. Ist kein Fehler aufgetreten, so beginnen nun die Kompiliervorgänge 'lc1' und 'lc2'. 'lc1' überprüft hauptsächlich die Syntax des Hauptprogramms und der geladenen Include-Files. 'lc2' generiert anschließend den Programmcode. Tritt beim Compiliervorgang 'lc1' ein Fehler auf, so wird auch hier der Ablauf des Batch-Files gestoppt und eine Fehlermeldung 'Compiler-Fehler' ausgegeben. Sind die Kompiliervorgänge ohne Fehler abgelaufen, beginnt das Programm mit dem Zusammenfügen, sprich 'Linken', der Bibliotheksmodule mit dem Programmcode. Eine Meldung teilt dem Benutzer anschließend mit, daß dieser Prozeß beendet ist. Danach kann das 'kompilierte' und 'gelinkte' Programm gestartet werden. Das Programm muß unter dem gewünschten Namen, mit angehängtem '.c' erstellt und abgespeichert worden sein, also beispielsweise 'test.c'. Nach dem Kompilier- und Linkvorgang steht der startbare Programmcode in der Datei 'test'. Dieser Programmcode läßt sich nun einfach durch Eintippen des Dateinamens starten. Es bietet sich auch noch die Möglichkeit an, eine '.info'-Datei zu kopieren, beispielsweise 'copy cli.info to test.info'. Dann können die Programme auch von der Workbench aus gestartet werden.

Für alle nachfolgenden Erklärungen möchten wir Sie bitten, alle Schritte direkt am Computer nachzuvollziehen, da Sie die Schritte dann leichter verstehen. Zu Beginn müssen Sie natürlich den Computer starten. Falls noch nicht geschehen, starten Sie das Preference-Programm und setzen Sie den CLI-Schalter auf 'ON', da nur in diesem Fall CLI zu verwenden ist. Anschließend können Sie Preference wieder verlassen (am besten mit 'Save'), da dann das CLI-Icon auch nach dem Einschalten des Computers erscheint. Nun starten Sie bitte CLI durch einen Doppelklick auf das CLI-Icon. Kurz darauf erscheint das CLI-Window. Ist es das einzige CLI-Window auf dem Bildschirm, so müßte das '1>'-Prompt darin erscheinen. Dahinter ist der Cursor zu erkennen. Nun können Sie sämtliche CLI-Befehle verwenden. Als Beispiel dafür

tippen Sie bitte 'dir' (mit anschließendem 'RETURN') ein. Sie sehen nun das Inhaltsverzeichniss der Hauptdiskette.

Nun wollen wir mit der Einführung in 'C' beginnen. C ähnelt in vielen Punkten den Programmiersprachen PASCAL und MODULA, weshalb PASCAL- und/oder MODULA-Programmierer keine Schwierigkeit haben dürften, auf C umzusteigen.

'C' wurde 1972 in den USA entwickelt, 1973/74 verbessert und anfangs vornehmlich unter dem Betriebssystem UNIX verwendet. Da diese Sprache möglichst flexibel sein sollte, wurden ihr nur sehr wenige Befehle fest implementiert.

Diese wenigen Befehle genügen allerdings, um alle programmkontrollierenden Funktionen durchführen zu können, zumal sich eine Vielzahl von Befehle in Include-Dateien oder in Bibliotheken befinden, die der Sprache C zu ihrer Leistungsfähigkeit und Flexibilität verhelfen.

C-Programme wirken leider teilweise sehr undurchsichtig, was aber durch das Einfügen von Kommentaren und Unterroutinen wett gemacht werden kann. Solche Kommentare klammert man mit '/*' und '*/' geklammert. Aufpassen sollten Sie auf solche 'Kleinigkeiten' wie Semikolons oder Kommata, da der Compiler oftmals solche Fehler nicht erkennt, sondern weiterkompiliert. Dies führt beim späteren Starten des Programms unweigerlich zum Absturz. Mit 'C' zu arbeiten heißt also korrekt und sauber arbeiten, sonst kann für nichts garantiert werden.

An dieser Stelle wollen wir auf die möglichen Datentypen eingehen, mit denen Variablen deklariert werden können:

Datentyp	**Wertebereich**		**Speicherlänge**
int	- -32768	bis 32767	- 2 BYTES
long int	- -2*10 hoch 9	bis 2*10 hoch 9	- 4 BYTES
unsigned int	- 0	bis 65535	- 2 BYTES
char	- -128	bis 127 (ASCII)	- 1 BYTE
unsigned char	- 0	bis 255	- 1 BYTE
FLOAT	- ±10 hoch -37	bis ±10 hoch 38	- 4 BYTES
DOUBLE	- ±10 hoch -307	bis ±10 hoch 308	- 8 BYTES
BYTE	- -128	bis 127	- 1 BYTE
UBYTE	- 0	bis 255	- 1 BYTE
WORD	- -32768	bis 32767	- 2 BYTES
UWORD	- 0	bis 65535	- 2 BYTES
LONG	- -2.15*10hoch9	bis 2.15*10 hoch 9	- 4 BYTES
ULONG	- 0	bis 4.3*10 hoch 9	- 4 BYTES

Sie werden später feststellen, daß dies einige Datentypen mehr sind, als beispielsweise beim MSC-Compiler für IBM-Systeme. Dies ist damit zu erklären, daß beim Amiga sehr oft Funktionen verwendet werden, die direkt BYTE, WORD oder LONG-WORD-Parameter verwenden. So sind auf dem Amiga einfach einige Datentypen hinzugekommen, die dies unterstützen. Allerdings sind es keine absolut neuen Typen, sondern über Präprozessoranweisungen entstanden. BYTE ist zum Beispiel der gleiche Datentyp wie char und UBYTE (unsigned BYTE) wie unsigned char. Die Präprozessoranweisungen für diese 'neuen' Datentypen stehen in der Include-Datei 'Exec/Typen'. Um diese Typen verwenden zu können, müssen Sie also den Präprozessorbefehl '#include <exec/types>' in Ihr Programm einbauen.

Die Hardware des Amiga ist von einer Vielzahl von leistungsstarken Software-Modulen umgeben. Durch diesen modularen Aufbau bieten sich ungeahnte Möglichkeiten. Das System wird flexibler und leistungsstärker, Module können hinzugefügt oder (falls notwendig) verändert werden.

Einen Teil dieser Amiga-System-Software-Module bilden die Libraries, zu deutsch (Software-)Bibliotheken. Das Amiga-System enthält bisher 16 Module. Hier eine Übersicht :

clist.lib: Enthält einige nützliche Routinen, die den Umgang und die Anwendung der Copper-Liste vereinfachen.

console.lib: Diese Library enthält Programme für den Umgang mit der Tastatur, der sogenannten Console.

diskfont.lib: Sie ermöglicht die Verwendung der verschiedenen Schrifttypen, die sich auf der Workbench-Diskette befinden.

dos.lib: Durch diese Library wird dem Amiga unter anderem der Zugriff auf die Diskette ermöglicht. Der Zugriff auf die Diskette ist dank dieser Library fast so einfach, wie von der Benutzerschnittstelle CLI aus.

exec.lib: Diese Library bildet den System-Kern des Amiga. Dieser Kern entscheidet beispielsweise, welche Tasks zum Laufen kommen (in der Computersprache bezeichnet man dies mit Scheduling) oder wieviel Speicherplatz für ein Programm bereitgestellt werden muß.

graphic.lib: Ohne Grafik geht heutzutage nichts mehr. Diese Library ist ein sehr leistungsstarkes und umfangreiches Bibliotheksmodul, dessen Funktionen unter anderem durch den direkten Zugriff auf den Blitter und Copper fantastische Geschwindigkeiten in punkto Grafik sowie Animation ermöglichen.

icon.lib: Hier sind verschiedene, durchaus nützliche Utilities für den Umgang mit den, von der Workbench her bekannten Icons enthalten. Es ist ebenfalls eine der wenigen Libraries, die sich auf der Workbench-Disk befinden.

info.lib: Diese Library wird dazu verwendet, Information über Dateien, Datei-Verzeichnisse oder ganze Disketten zu bekommen.

Sie befindet sich auf der Workbench-Diskette und wird nur selten verwendet.

intuition.lib: Sie ist eine der wichtigsten Libraries des Amiga. Ohne sie wäre keine Bedienung mit der Maus oder die einfache Handhabung von Menüs denkbar.

janus.lib: Dies ist bisher das letzte Bibliotheks-Modul, das dem Amiga beigefügt wurde. Es befindet sich ebenfalls auf der Diskette und wird zur Steuerung der Side-Car-Hardware benötigt.

layers.lib: In diesem Bibliotheks-Modul sind Routinen enthalten, die dem Anwender beispielsweise das Handling von überlappenden Display-Elementen erleichtern.

mathffp.lib: Mit diesem sogenannten FFP-Basic-Mathematik-Library können einfache mathematische Aufgaben, wie z.B. die Multiplikation oder Division, gelöst werden.

mathieeedoubbas.lib: Dies ist das erweiterte FFP-Basic-Mathematik-Library. Es befindet sich auf der Workbench-Diskette und enthält eine Vielzahl von mathematischen Funktionen, die Zahlen im IEEE-Standard mit doppelter Genauigkeit verarbeiten.

mathtrans.lib: Für schwierigere mathematische Aufgaben, bei denen Funktionen wie arcsin, arccos usw. Verwendung finden, enthält dieses Bibliotheks-Modul genügend Befehle. Da diese Funktionen nicht ständig verwendet werden, ist diese Library auf der Worbench-Disk enthalten.

timer.lib: Wenn Sie zeitlich im Bilde sein wollen, bietet sich die Verwendung dieses Libraries an. Leider kann beim Amiga 1000 nur die Software-Uhr angesprochen werden. Bei den Versionen 500 und 2000 ist diese Uhr jedoch batteriegepuffert.

translator.lib: Sie hat die Aufgabe, Sätze für die Sprachausgabe vorzubereiten, die in englisch verfaßt sind. Es findet kaum Verwendung und ist deshalb auf der Systemdisk enthalten.

Je nach Art des Programms, das der Programmierer entwickeln will, muß er selbständig entscheiden, welche Bibliotheks-Module er benötigt. Sicherlich werden Sie nun denken, je mehr Libraries verwendet werden, desto besser wird das Programm. Im Gegenteil! Für sehr gute Programme reichen teilweise schon 2 bis 3 Libraries aus.

Beim Umgang mit den Libraries müssen bestimmte Regeln eingehalten werden, damit die jeweiligen Funktionen ansprechbar sind. So hat es natürlich keinen Zweck, Funktionen einer Library aufzurufen, wenn sie nicht vorher geöffnet wurde.

Bevor jedoch die jeweilige Library geöffnet wird, muß der 'Basis' des Library ein Zeiger zugewiesen werden, hier am Beispiel des Intuition Library demonstriert, der von OpenLibrary zurückgegeben wird:

```
IntuitionBase = (struct IntuitionBase *)
  OpenLibrary("intuition.library", 0);
```

Anschließend enthält IntuitionBase die Einsprungaddresse der Intuition-Library. Enthält diese Variable den Wert 'NULL', war es nicht möglich, die Library zu öffnen.

Ist der Wert ungleich 'NULL', verlief alles normal und die Library konnte geöffnet werden.

Wenn Sie eine Library geöffnet haben, muß sie natürlich auch wieder geschlossen werden:

```
CloseLibrary(IntuitionBase);
```

Hier zum besseren Verständnis nochmal ein Beispiel:

```
/* Öffnen und Schließen eines Bibliotheksmodules */
#include <graphics/gfxbase>
.
.
struct GfxBase *GfxBase;  /* Zeiger für die Einsprungadresse */
.                                 /* deklarieren */
.
main()
{
.
GfxBase = (struct GfxBase *)
  OpenLibrary("graphics.library", 0); /* Library öffnen */
if (GfxBase == NULL)
  {
   printf("Öffnen des graphics.library nicht möglich !\n");
   exit(FALSE);
  }
/*
        Hier das jeweilige Programm eintragen
*/
/* Zum Schluß Bibliotheks-Module schließen */
CloseLibrary(GfxBase);
}
```

IntuitionBase und GfxBase dürfen mit

```
struct IntuitionBase *IntuitionBase;
```

bzw.

```
struct GfxBase *GfxBase;
```

deklariert werden, da sie die Einsprungaddressen der Intuition- und der Graphics-Library darstellen. Für alle anderen Libraries gilt folgende Deklaration:

```
Beispiel Diskfont-Library
.
.
ULONG DiskfontBase;
.
.
main()
{
.
DiskfontBase = OpenLibrary("diskfont.library",0);
if (DiskfontBase == NULL)
  {
   printf("Öffnen des diskfont.library nicht möglich !\n");
   exit(FALSE);
  }
/*
        Hier das jeweilige Programm eintragen
*/
/* Zum Schluß Bibliotheks-Module schließen */
CloseLibrary(DiskfontBase);
}
```

Neben den Libraries, die für den Programmierer Erleichterungen und für das System eine große Flexibilität darstellen, stehen dem Programmierer weitere Hilfsmittel zur Verfügung: die Devices.

Devices, zu deutsch Vorrichtungen, sind die Bindeglieder zwischen der (externen) Hardware und der Software des Amiga. Durch sie können Daten zur Hardware gesendet oder von ihr empfangen werden. Somit ist, z.B. durch das Verändern von Parametern der Trackdisk-Device, das Lesen von fremden Diskettenformaten wie IBM oder Apple-Format möglich.

Der Amiga enthält 17 verschiedene Devices, die sich um die Vorrichtungen, wie Tastatur, serielle und parallele Schnittstellen und einiges mehr kümmern. Nicht alle

werden ständig benötigt, sondern befinden sich im 'Devs'-Directory auf der Workbench-Disk.

Die Devices des Amiga im Überblick :

audio.device: Mit ihr wird der 'Sound' des Amiga gesteuert. Je nach Belieben richtet sie die 4 Audio-Kanäle des Amiga ein, bestimmt die Amplitute des Tons und vieles mehr.

bootblock.device: Testet, ob es sich um eine Kickstart- oder um eine DOS-Diskette handelt. Bei den neuen Amigas ist diese Vorrichtung weggefallen, da bei ihnen keine Kickstart-Diskette mehr erforderlich ist.

clipboard.device: Wird benötigt, um Daten zwischen zwei Anwendungen zu transferieren. Da dies nicht häufig vorkommt, befindet sich diese Device auf der Workbench-Disk.

console.device: Regelt die Ein- und Ausgabe des Systems über die Tastatur und den Bildschirm.

gameport.device: Übernimmt die Steuerung der Ein- und Ausgabe über die Game-Ports 1 und 2.

input.device: Diese Device regelt die gesamte Ein- und Ausgabe des Amiga. Es ist eine Kombination aus timer-, gamport- und keyboard.device.

inputevent.device: Inputevent.device erfaßt die Ereigniseingaben, wie z.B. Gadgets.

jdisk.device: Dies ist die neuste Device des Amiga. Sie übernimmt die Steuerung der Harddisk des Amiga, die sich auf der IBM-PC-kompatiblen Seite des Amiga2000 oder im SideCar befindet. Da sie sehr neu ist, befindet sie sich ebenfalls auf der Workbench-Disk.

keyboard.device: Hiermit wird der Zugriff auf die Tastatur des Amiga gesteuert.

keymap.device: Damit kann die Belegung der Tastatur verändert werden.

narrator.device: Narrator.device ist für die Steuerung der Sprachausgabe notwendig. Da sie nicht ständig benötigt wird, befindet sie sich auf Workbench-Disk.

parallel.device: Hiermit kann der Parallelport gesteuert werden. Diese Device befindet sich ebenfalls auf der Worbench-Disk.

printer.device: Diese Device dient zur Kommando-Steuerung des Druckers, um z.B. einen Wagenvorlauf des Druckers zu bewirken. Printer.device befindet sich ebenfalls auf der Workbench-Disk.

prtbase.device: Prtbase.device übernimmt die Datendefinition der printer.device.

serial.device: Diese dient der Deklaration des seriellen Ports des Amiga. Sie befindet sich ebenfalls auf der Workbench-Disk.

timer.device: Mittels Timer.device kann auf die Systemzeit zugegriffen werden.

trackdisk.device: Diese Device kontrolliert die Floppies des Amiga. Sie übernimmt Funktionen, wie das Lesen und Schreiben von Daten und einiges mehr.

Um mit einer Device arbeiten zu können, muß sie geöffnet werden. Dies geschieht mit

```
printerPort = CreatePort("printer.port",0);
```

wobei printerPort zurückgegeben wird.

Danach muß die Device geöffnet werden. In diesem Fall "printer.device":

```
fehler = OpenDevice("printer.device",0,&request,0);
```

Wenn Fehler ungleich 0 ist, konnte die Device nicht geöffnet werden. &request ist der Pointer auf eine Structure der jeweiligen Device, die bestimmte 'Routinen' wie z.B. das Drucken eines Screens enthalten oder auf die allgemeine Ein-/Ausgabe-Structure von EXEC.

Nachdem Device und Port geöffnet sind, kann die benötigte Ein- und Ausgabe-Structure initialisiert werden.

Nach der Initialisierung wird die jeweilige 'Funktion', wie z.B. das Drucken eines Textes, mit

```
DoIO(&request);
```

gestartet.

&request ist der Pointer auf die Ein- und Ausgabe-Structure der jeweiligen 'Funktion'.

Nachdem die Ein- und Ausgabe beendet ist, muß der Port und die Device wieder geschlossen werden. Dies kann mit

```
DeletePort(printerPort);
CloseDevice(&prefrequest);
```

Dieses System klingt für Sie höchstwahrscheinlich sehr umständlich, hat aber sehr viele Vorzüge. Da dieses Thema aber schon eher für etwas fortgeschrittene Programmierer interessant ist, möchten wir hier nicht näher auf dieses Thema eingehen, sondern verweisen Sie auf das Amiga-Programmierhandbuch, das ebenfalls im Markt&Technik-Verlag erschienen ist, da es sich genauestens mit diesem Themenkomplex beschäftigt.

2 Warum C?

Warum ausgerechnet C? Warum nicht BASIC, COBOL, Fortran, PL/1, Forth, Prolog, Lisp etc.? Diese Diskussion artet bisweilen in eine Glaubensdiskussion aus. In der Tat glaube ich nicht, daß es eindeutige Argumente für die eine und gegen alle anderen höheren Programmiersprachen gibt. Deshalb sollte der Leser meine folgenden Argumente kritisch bedenken: eine andere bevorzugte Programmiersprache ist durchaus zugelassen. Ich bin kein orthodoxer Prediger der C-Gemeinde.

Zunächst: C ist eine **Compilersprache** und keine **Interpretersprache**. Damit sind BASIC, Lisp, Prolog und Forth aus dem Rennen. Interpretersprachen haben unbestreitbare Vorteile: die Programme können interaktiv ausgetestet werden. Man braucht den üblichen Zyklus Editor – Compiler – Linker – Ausführung nicht zu durchlaufen, da Interpretersprachen zumindest den Linker entbehrlich machen und oft auch über einen integrierten Editor (s. BASIC und Forth) verfügen.

Dafür aber haben Interpretersprachen zwei entscheidende Nachteile: der Interpreter ist immer (unsichtbar) Bestandteil des Programms – er ist und bleibt geladen, weil die Programme nicht auf die Ebene von Maschineninstruktionen übersetzt werden, sondern vom Interpreter erst 'interpretiert' werden müssen, bevor sie ausgeführt werden können. Der Nachteil ist doppelt:

- hoher Platzbedarf: auch das kleinste Programm kann nicht kleiner sein als der Interpreter, unter dem es abläuft
- Geschwindigkeit: interpretierte Programme laufen langsamer ab als kompilierte Programme.

Besonders der Geschwindigkeitsvorteil spricht für die kompilierten Programme – deutlich zu erkennen an der Tatsache, daß es für einige BASIC-Interpreter auch Compiler gibt, die linkfähigen Code erzeugen, der dann deutlich schneller abläuft als das interpretierte Programm. Im Bereich der FORTH-Gemeinde (oder sollte man

besser Sekte sagen?) geht das Gerücht um, FORTH sei unter Umständen sogar schneller als Assembler. Dies ist schlicht und einfach falsch. Jede von mir bisher geprüfte FORTH-Implementation war mir zu langsam. Aber vielleicht muß ich mir meine Problemstellungen etwas besser aussuchen...

Um das nochmal zu sagen: ich habe nichts gegen Interpreter – im Gegenteil: Fehler suchen ist eine der Hauptbeschäftigungen professioneller Programmierer, und das wird in einer interpretierenden Umgebung besonders leicht gemacht. Aber: Geschwindigkeit ist Geld und den Kunden (den Anwender) interessiert nicht, ob ein Programm besonders einfach 'entwanzt' werden kann. Er erwartet fehlerfreie Programme. Zudem gibt es Programme, die schlicht und einfach nicht in einer Interpreterumgebung getestet werden können: etwa eine zeitkritische Interruptserviceroutine, effiziente Videoroutinen oder Programme, die gezielt Speicherbereiche verändern, die der Interpreter etwas vorschnell für sich reserviert hat.

Bleiben also die Compilersprachen. Warum aber C statt Fortran, PL/1, COBOL, Pascal etc?. Auch hier ist die Entscheidung nicht einfach. Um ehrlich zu sein: nur PL/1 traue ich es zu, eine ernsthafte Konkurrenz zu C zu sein. PL/1 aber ist praktisch tot, da es keine billigen Implementierungen für Personal Computer gibt. Zudem sind die meisten Implementierungen nur Untermengen (Subset G) der ursprünglichen Sprachdefinition, wie sie auf IBM-Großrechnern realisiert ist. Die wichtigsten Argumente für C sind:

- weite Verbreitung: Implementierungen auf Personal Computern unter CP/M über Minicomputer (alle UNIX-Rechner) bis zu Großrechnern (z.B. VAX der Firma DEC).
- de facto Normierung: nahezu alle guten Implementierungen orientieren sich am Kernighan & Ritchie (= UNIX) Standard. Eine ANSI-Norm wird vorbereitet, die jedoch den de facto Standard als Untermenge enthält.
- Austauschbarkeit: da C auf einer Vielfalt von Computern existiert und überdies faktisch normiert ist, sind die meisten C-Programme portabel, also vom Entwicklungsrechner auf andere Rechner übertragbar (portierbar).
- Übersichtlichkeit: C besteht im Kern nur aus wenigen Sprachelementen und ist deshalb leicht erlernbar.
- Vollständigkeit: C weist alle praktisch wichtigen Eigenschaften auf, besonders die Eigenschaft, eine strukturierte (block- und prozedurorientierte) Sprache zu sein. Die Fans der strukturierten Programmierung können zufrieden sein.
- Realismus: C ist von Praktikern für Praktiker geschrieben.
- Flexibilität: Der Einsatz von Standardmitteln in C erspart in vielen kritischen Fällen Assembler. Spracherweiterungen, die – wie man in Pascal und besonders BASIC beobachten kann – zur Dialektbildung führen, sind für effizientes Arbeiten nicht erforderlich.

- Universalität: C ist – entgegen anderslautenden Gerüchten – nicht für spezielle Aufgaben konzipiert. C ist im Gegenteil eine universelle Programmiersprache. Die große Zahl der professionellen Anwendungen vom Betriebssystemkern über die Textverarbeitung bis zur Lohnbuchhaltung zeigt dies.

Es soll nicht verschwiegen werden, daß C auch Nachteile hat. Als Hauptargument wird immer angeführt: C-Programme sind schlecht lesbar. Dies ist weitgehend korrekt, wenn der de facto Programmierstil vieler C-Programmierer untersucht wird. Daß C-Programme schlecht lesbar sind, hat drei Gründe: zum einem orientieren sich viele Programmierer an Kernighan & Ritchie, die nicht viel zur Lesbarkeit von Programmen beitrugen, weil sie (wie im UNIX-Stil üblich) sehr kompakte Programme schrieben und nicht die glücklichsten Schreibkonventionen ausgesucht haben. Der zweite Grund liegt sicher darin, daß C sehr lange die Domäne der UNIX-Gurus war – und welcher Guru will sich schon in die Karten gucken lassen? Der dritte Grund liegt darin, daß C-Programme oftmals komplexere und ungewohntere Probleme bewältigen als in anderen Programmiersprachen üblich. Daß C-Programme schlecht lesbar sind liegt also dann daran, daß die zugrundeliegende Problemstellung nicht trivial ist.

Ich habe dem Problem der Lesbarkeit von C-Programmen ein eigenes Kapitel gewidmet und bitte Sie, dieses Problem ernst zu nehmen. Das Argument, das aus meiner Sicht am meisten gegen C spricht, ist das Problem der **Fehleranfälligkeit**. C ist eine sehr liberale Sprache, die mehr syntaktische Freiheiten einräumt als andere Programmiersprachen. Das führt zu katastrophalen Fehlern selbst bei erfahrenen C-Programmierern. Für den C-Anfänger heißt das: die Fehlersuche dauert länger als in anderen Programmiersprachen.

3 Erste C-Programme

Ich steige ohne große Vorreden gleich in das erste C-Programm ein. Dieses Programm liest einen Satz (eine Zeile von Zeichen) ein und zählt, wieviele Leerstellen (Blanks) der Satz enthält. Das Ergebnis wird ausgegeben. Natürlich ist das ein nicht gerade sehr aufregendes Beispielprogramm. Sie sollten sich aber auf jeden Fall die Zeit nehmen, das Beispiel anzusehen und an Ihrem Computer einzutippen. Wenn Sie dieses Beispiel verstehen, kennen Sie eigentlich schon die wichtigsten Elemente der Sprache C. Nach der Lektüre dieses Kapitels sollten Sie auch in der Lage sein, Ihren C-Compiler samt Linker aufzurufen und somit ein einfaches lauffähiges Programm zu erstellen.

3.1 Programmabdruck

```
/****************************************

     Erstes sehr einfaches C-Programm
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Das Programm zaehlt die Blanks in einem Satz

****************************************/

#include "stdio.h"

#define  ENDLOS  1                                    /* 040 */
                                                      /* 050 */
```

```
char satz[80];                                          /* 060 */
                                                        /* 070 */
main()                                                  /* 080 */
{                                                       /* 090 */
  int anzbl, n;                                         /* 100 */
                                                        /* 110 */
  while (ENDLOS)                                        /* 120 */
    {
     printf("\n%s","Bitte einen Satz eingeben:");
     gets(satz);
     if (satz[0] == 0) break;
     anzbl = 0;
     for (n=0; n < 80; n++)
       {
        if (satz[n] == 0) break;
        if (satz[n] == ' ') anzbl++;
       }
     printf("\nDer Satz hat %d Blanks",anzbl);
    }  /* while ENDLOS */
  printf("\n%s\n","--- Programmende ---");
}   /* end main */
```

3.2 Kommentare

Vielleicht fällt zuerst auf, daß einige Zeilen am rechten Rand numeriert sind – keine Angst: das ist nicht nötig und dient hier nur dazu, sich auf die Zeilen beziehen zu können. Zeilennummern wie in BASIC oder gar wie in alten Lochkartenzeiten sind in C nicht nötig.

Das Programm besteht aus vier Teilen, die in der Länge unterschiedlich sind: dem Kommentarteil, der die Gesamtfunktion erklärt, dem Präprozessorteil, der Deklaration der externen Daten und dem Prozedurteil. Sie werden sehen, daß sich diese Einteilung als vorteilhaft erweist. Dieses Programm besteht nur aus einer Prozedur. Größere Programme umfassen meist mehrere Prozeduren.

Zeile 10: Diese Zeile und die folgende Zeile besteht aus Kommentaren. Kommentare beginnen mit der Zeichenkombination /* und enden mit */. Auch die schon erwähnte Zeilennumerierung ist eigentlich ein Kommentar. Kommentare können sich über mehrere Zeilen erstrecken und an beliebiger Stelle (Spalte) anfangen und aufhören.

Zeile 30: Diese und die folgende Zeile enthalten Präprozessorbefehle. Diese Befehle beginnen immer in Spalte 1 einer Zeile mit dem Zeichen #. Hier sollen Sie sich nur merken, daß die Zeile 20 *(#include <stdio.h>)* praktisch immer benötigt wird. Die Zeile 30 definiert ein Konstantensymbol: alle Vorkommen von *ENDLOS* im Programmtext sollen durch die 1 ersetzt werden. Wenn das Programm tatsächlich läuft, steht in Zeile 120 nicht *while (ENDLOS)* sondern *while* (1).

Zeile 50: Überall im Programm können Leerzeilen erscheinen. Verwenden Sie Leerzeilen zur Steigerung der Lesbarkeit eines Programms. Hier soll die Leerzeile den

Präprozessorteil (Zeilen 30-40) vom Deklarationsteil (Zeile 60) optisch trennen. Nötig ist diese Zeile wie gesagt nicht!

Zeile 60: Eine **externe** Datendeklaration. *satz* wird deklariert als ein Array aus 80 Elementen vom Typ *char*. Für die Angabe der Anzahl Elemente in einem Array werden eckige Klammern verwendet. Die Deklaration ist ein normaler C-Befehl und wird deshalb mit einem Semikolon (Strichpunkt) beendet. Die Deklaration heißt hier extern, weil sie nicht innerhalb einer Prozedur (diese beginnt erst in Zeile 80) erfolgt, sondern außerhalb. Eine lokale (interne) Deklaration findet sich in Zeile 100. Externe Daten können von allen Routinen eines Programms verwendet werden, lokale nur innerhalb einer Prozedur oder eines Blocks.

Zeile 80: Hier beginnt die Prozedur *main*(), also das Hauptprogramm. Die Bezeichnung 'Hauptprogramm' ist etwas hochtrabend, weil gar keine anderen Prozeduren definiert sind. Außer in sehr speziellen Anwendungen hat ein C-Programm immer eine (und nur eine!) *main*-Routine. Das Klammerpaar () hinter *main* kennzeichnet den Start einer Prozedur. In komplexeren Programmen können zwischen den Klammern auch Parameter erscheinen. Übrigens gibt es in C formal keinen Unterschied zwischen *main* und beliebigen Prozeduren. *main* ist kein C-Befehl, sondern nur der Name der Prozedur. Deshalb steht hier kein Semikolon.

Zeile 90: Die geschweifte öffnende Klammer besagt, daß jetzt der inhaltliche Teil der Prozedur *main* beginnt. Die dazu gehörige schließende Klammer steht in Zeile 260. Geschweifte Klammern haben in C die Funktion, zusammengehörige Programmzeilen zu kennzeichnen. PASCAL-geübte Leser erkennen sofort die Parallele zu *begin* und *end*. Man spricht hier auch von **Blöcken**.

Zeile 100: Hier werden zwei lokale Variablen vom Typ *int* (= integer) definiert, nämlich *anzbl* und *n*. Die Variablen heißen 'lokal', weil sie innerhalb der Prozedur *main* deklariert werden und somit nur für *main* gelten. Eine eventuell vorhandene andere Prozedur kann die gleiche Deklaration aufweisen und damit gleichlautende Variablen verwenden - sie sind dennoch nicht identisch. Der Typ *int* sagt übrigens, daß die Variablen ganzzahlige, vorzeichenbehaftete Werte aufnehmen können. Typische Werte sind also z.B. **1, 100,1000, -1, -100, -1000** nicht aber **10.5** oder **-100.87**.

Zeile 120: Das ist der Anfang einer Endlosschleife. Die Zeile kann so übersetzt werden: 'führe die Programmzeilen zwischen Zeile 130 (öffende Klammer) und 240 (schließende Klammer) endlos oft aus'. *while* heißt eigentlich 'solange'. Also kann ich auch übersetzen: 'führe die Schleife aus, solange *ENDLOS* nicht gleich 0 ist'. Da *ENDLOS* als Konstante 1 (s. Zeile 40) definiert ist, kann *ENDLOS* nie 0 werden - die Schleife ist also endlos. Schleifen werden wie in diesem Beispiel durch ein Schleifenkennwort (hier *while*) eingeleitet und weisen dann einen Schleifenkörper auf, der hier zwischen den geschweiften Klammern steht. Dieser Schleifenkörper stellt einen zusammengesetzten Befehl dar, auch Block genannt. Schleifenkonstrukte haben unterschiedliche Start- und Abbruchbedingungen. Bei der *while*-Schleife wird vor jedem Schleifendurchlauf geprüft, ob der hinter *while* folgende Ausdruck noch wahr (jeder

ganzzahlige Wert ungleich 0 gilt als wahr) ist. Ist dies nicht der Fall, so wird die Schleife abgebrochen.

Zeile 140: Jetzt passiert etwas am Bildschirm: der Text 'Bitte einen Satz:' wird ausgegeben. *printf*() ist eine Funktion der C-Bibliothek (library), gehört also nicht zum engeren C-Sprachumfang. *printf*() hat in diesem Beispiel zwei Argumente: den String "*n%s*" und den String **"Bitte einen Satz:"**. Beide Argumente werden durch Komma voneinander getrennt. Den ersten String nennt man auch Formatanweisung. *n* steht hier übrigens für 'erzeuge zuerst einen Zeilenvorschub' und resultiert in der Ausgabe eines Linefeeds *(LF)*. Die meisten Systeme ergänzen das *LF* zu *CR* (Carriage Return, Wagenrücklauf) + *LF*. *%s* heißt: Gib das nachfolgende Argument (den zweiten String) als String aus. In Zeile 230 ist ein zweites Beispiel, bei dem das zweite Argument als Zahl ausgegeben wird. Die genauere Funktion von *printf*() erläutere ich später. Wie in *main*() (das keine Argumente hat) stehen die Argumente der Funktion *printf*() zwischen runden Klammern. *printf*() ist eine Funktion, die nicht im Programm selbst definiert wird, sondern in der C-Bibliothek bereits vorhanden ist. Der Linker sorgt später dafür, daß der zu *printf*() gehörige Objektcode ins Programm eingebunden wird.

Zeile 150: *gets*() ist ebenfalls eine Funktion der C-Bibliothek. Sie sorgt dafür, daß eine Zeile von der Tastatur eingelesen wird und im Array *satz* abgespeichert wird. Eine Zeile ist übrigens eine Folge von Zeichen, die durch ein *CR* bei der Eingabe abgeschlossen wird. Das *CR* wird von *gets*() nicht übernommen, sondern in eine binäre 0 umgewandelt. In der C-Terminologie sagt man: *gets*() liefert einen **String**. Strings sind Folgen von Zeichen, die durch eine binäre Null (Code 0x00, nicht mit der ASCII Null mit Code 0x30 verwechseln) abgeschlossen werden. Während der Eingabe des Satzes dürfen Sie sich verschreiben und mit Backspace korrigieren – die Funktion *gets*() erlaubt Ihnen das. Ergebnis von *gets*() in *satz* ist die gereinigte, endgültige Eingabe als String. In Zeile 150 werden ebenfalls Strings verwendet, allerdings Stringkonstanten, die zwischen doppelten Hochkommata erscheinen. Strings können in *char*-Arrays abgelegt werden. Das erledigt hier *gets*().

Zeile 160: Hier wird abgefragt, ob die Eingabe leer war, d.h. ob der Benutzer des Programms nur die Return-Taste gedrückt hat, ohne vorher etwas einzugeben. Diese leere Eingabe betrachtet das Programm als Wunsch, die Schleife zu beenden: mit *break* kann man die aktuelle (die jeweils innerste) Schleife verlassen. Die leere Eingabe (= der leere String) äußert sich in einer binären Null im Element 0 des Arrays *satz*. In dieser Zeile sind gleich zwei äußerst wichtige C-Eigenschaften zu notieren: das erste Element eines Arrays in C ist immer über den Index 0 ansprechbar – nicht etwa über 1 wie in Fortran oder anderen Programmiersprachen. Die zweite C-Eigenheit besteht in der Notation der Abfrage auf Gleichheit: hier wird == verwendet, also ein doppeltes Gleichheitszeichen. Zum Vergleich sehen Sie sich bitte Zeile 170 an: das einfache Gleichheitszeichen dient der Zuweisung eines Wertes, das doppelte Gleichheitszeichen der Abfrage auf Gleichheit. Die Verwechslung beider C-Operatoren ist so ziemlich der beliebteste Fehler, den C-Programmierer begehen.

Zeile 170: Hier beginnt die Auswertungslogik, also das Zählen der Blanks im Inputsatz. Deshalb wird die Variable *anzbl* auf 0 gesetzt. Sie soll die Zahl der Blanks aufnehmen und muß deshalb initialisiert werden. Das Zeichen = ist der Zuweisungsoperator. Links von = steht fast immer eine Variable, rechts davon eine Variable, eine Konstante oder eine Funktion. Ich sage 'fast immer', weil es in C Operatoren gibt, die gleichsam Variablen erzeugen. Mehr davon später. Bitte beachten Sie nochmals den Unterschied zwischen == und = in Zeile 160 und 170.

Zeile 180: Hier beginnt eine zweite Schleife, deren Ende die Zeile 220 bildet. Diese Schleife steht innerhalb der *while*-Schleife von Zeile 120, die ja erst in Zeile 240 endet. Die *for*-Schleife ist etwas komplizierter als die *while*-Schleife. Sie faßt innerhalb der runden Klammern drei Befehle zusammen: einen Initialisierungsbefehl für die Schleifenvariable n, eine (verkürzt notierte) Abfrage *n < 80* und das Erhöhen der Schleifenvariablen *n* notiert als *n++*. Jeder Einzelbefehl außer dem letzten muß mit einem Semikolon abgeschlossen werden. Der Initialisierungsbefehl *n* = 0 setzt die Variable n auf 0, damit sie den erforderlichen Startwert annimmt. Der zweite Teilbefehl gibt die Bedingung an, wie lange die Schleife fortgesetzt werden darf; wenn der Ausdruck *n* < 80 wahr ist, wird die Schleife weiter fortgesetzt, wenn er falsch ist (also *n >= 80* wahr ist), wird die Schleife abgebrochen. Im Gegensatz zu den meisten anderen Programmiersprachen wird hier also nicht direkt auf Abbruch geprüft, sondern eher auf Erlaubnis zum Fortsetzen der Schleife. Der dritte Teilbefehl *n++* ist auch so eine typische C-Notation. ++ ist ein Operator, der die vorangehende Variable um 1 erhöht. Genauso legal ist hier ein Ausdruck wie *n = n + 1*. Wenn ich die *for*-Schleife in eine *while*-Schleife übersetze, wird ihre Bedeutung vollends klar:

```
/* Uebersetzung for in while */
n = 0;
while (n < 80)
{
   /* Hier wird irgend etwas getan */
   n++;       /* gleich n = n + 1; */
}
```

Bitte fügen Sie unmittelbar hinter der runden Klammer in Zeile 180 kein Semikolon ein – sonst tut die Schleife nichts außer der Erhöhung der Schleifenvariablen. Übrigens ist es nicht erforderlich, Schleifenvariablen im Sinne von BASIC oder Pascal oder Fortran zu verwenden: Sie können in diesen drei Teilbefehlen alles Mögliche tun. Wenn Sie *for (;;)* schreiben, dann haben Sie eine andere Variante der Endlosschleife ausprobiert.

Zeile 200: Die Zeile prüft, ob wir schon auf der Endemarke des eingelesenen Strings angekommen sind. Die Endemarke ist in C immer die binäre Null. Mit *break* wird wieder die aktuelle Schleife – also die *for*-Schleife – verlassen. Die *for*-Schleife hat also in Wahrheit zwei Abbruchbedingungen: entweder *n* ist >= *80* oder die Endemarke des Strings wird erreicht.

Zeile 210: Hier wird geprüft, ob das aktuelle Zeichen im Array *satz* an der Position *n* ein Blank ist. Wenn dies der Fall ist, wird die Variable *anzbl* erhöht mit der gleichen seltsamen Schreibweise, die schon in Zeile 180 aufgetaucht ist. Legitim ist hier auch *anzbl* = *anzbl* + *1*. Es gibt sogar eine weitere Möglichkeit, nämlich *anzbl* += *1*. Wenn Sie in den Zeilen 200 und 210 ein = statt eines == verwenden, werden Sie keine Fehlermeldung erhalten, weil auch dies syntaktisch erlaubt ist. Also achten Sie bitte genau auf den Bedeutungsunterschied von Zuweisungsoperator = und Gleichheitsoperator ==. Beachtenswert ist auch der Unterschied in der Schreibweise für Stringkonstanten (Begrenzungszeichen ist die doppelte Anführung) und für Zeichenkonstanten (Begrenzungszeichen ist der Apostroph). Auf den inhaltlichen Unterschied komme ich etwas später zu sprechen.

Zeile 220: Die *for*-Schleife wird hier beendet.

Zeile 230: Für jeden eingegebenen Satz wird hier gemeldet, wieviele Blanks er hat. *\n* steht wieder für Ausgabe eines Linefeeds (das die Ausgabe eines CR nach sich zieht). %d heißt: der auf den Formatstring folgende Parameter soll als dezimale Zahl ausgegeben werden. Die Zahl wird genau dort eingefügt, wo das %d erscheint, d.h. die erzeugte Ausgabe könnte z.B. so aussehen: 'Der Satz hat 11 Blanks'. Wenn Sie genau hinsehen, erkennen Sie, daß *printf()* den gleichen Aufbau hat wie in Zeile 140: zuerst erscheint der Formatierungsstring, dann erscheinen soviele Argumente, wie Formatierungsanweisungen (die beginnen mit %) im Formatierungsstring erscheinen.

Zeile 240: Die Zeile beendet die *while*-Schleife. Bei längeren Schleifen und besonders bei Schachtelung von Schleifen (wie hier) kommt es der Lesbarkeit zugute, wenn die schließenden Schleifenklammern von einem Kommentar begleitet werden, der angibt, welche Schleife jetzt eigentlich zugemacht wird.

Zeile 250: Nichts Neues

Zeile 260: Hier wird das Hauptprogramm und gleichzeitig die einzige Routine beendet. Wenn die Routine *main* heißt, wird in C übrigens automatisch ein Rücksprung ins Betriebssystem veranlaßt. Ein *return*-Befehl ist nicht nötig, schadet jedoch nicht. Auch ein Aufruf von *exit*() (s. Kapitel über die C-Bibliothek) wäre möglich.

3.3 Zusammenfassung

Was Sie jetzt schon über typische C-Programmierung wissen sollten, ist folgendes:

- C-Programme bestehen aus drei Sorten von Zeilen: Kommentaren, Präprozessorbefehlen und C-Befehlen.
- Kommentare stehen zwischen /* und */.
- Präprozessorbefehle haben in Spalte 1 der Zeile ein #. Sie weisen am Zeilenende normalerweise kein Semikolon auf.

- C-Befehle enden mit Semikolon. Sie können an beliebiger Stelle innerhalb einer Zeile beginnen. Bei den C-Befehlen sollten Sie unterscheiden zwischen Deklarationen und ausführbaren Befehlen.
- Es gibt externe und lokale Deklarationen in C. Lokale Deklarationen gelten nur innerhalb des aktuellen Blocks, der durch geschweifte Klammern markiert wird.
- Zuweisungsoperator = und Vergleichsoperator == sind sehr leicht zu verwechseln.
- Es gibt mehrere Typen von Schleifen, u.a. *while*-Schleifen und *for*-Schleifen. *for*-Schleifen sind eine etwas kompaktere Notation von *while*-Schleifen.
- Arrays in C beginnen ab der Indexposition 0.
- Mit *break* können Sie die jeweils aktuelle Schleife verlassen.

3.4 Kompilieren, Linken, Starten

Ein Programm hat verschiedene Lebensphasen:

- Die Erfassungsphase – auch Vorcompilezeit genannt.
- Die Compilephase (Compilezeit). Die Compilephase ist mehrphasig und umfaßt u.a. den Präprozessorlauf, die Syntaxprüfung, die Codegenerierung und die Codeoptimierung.
- Die Linkphase (Linkzeit)
- Die Ablaufphase (Laufzeit)

Zunächst einmal muß man das Programm mit einem Editor eintippen und in einer Datei ablegen. Üblicherweise haben Programmdateien für C das Suffix (Postfix, Extension) '.C' oder '.c'. Bei mir heißt dieses Programm 'cprog1.c'. Im obigen C-Text sollten Sie sich strikt an die vorgegebene Schreibweise halten. In C-Programmen haben Groß- und Kleinschreibung unterschiedliche Bedeutung.

Der nächste Schritt – das Programm sollte jetzt fertig eingetippt sein – besteht darin, das Programm zu kompilieren. Der C-Compiler ist ein Programm, das aus dem Quellcode (Sourcecode), also Ihrem soeben eingetippten Programm, einen übersetzten Objektcode erzeugt. Aus der Datei mit dem Suffix '.C' oder '.c' wird meist ein Programm mit dem Suffix '.O' oder '.o' erzeugt. Es gibt auch Compiler, die zunächst Assemblercode (Suffix '.ASM' oder ähnlich) erzeugen, der dann zuerst noch assembliert werden muß. Aber auch dann entsteht ein File mit dem Suffix '.O'. Dies ist noch nicht das ausführbare Programm! Wie schon erwähnt, kann auch der Compilelauf aus mehreren Einzelläufen bestehen.

Jetzt muß das Programm gelinkt werden. Der Linker erzeugt aus der '.o'-Datei ein ausführbares Programm, das in Amiga-DOS kein Suffix hat. Um ein ausführbares Programm zu erzeugen, braucht der Linker weitere Hilfsprogramme, die er einer

oder mehreren Bibliotheken (libraries) entnimmt. Diese Bibliotheken liefern Routinen für die Einbettung des Programms in das jeweilige Betriebssystem und für die Ein-/Ausgabe.

3.5 Etwas C-mäßiger formuliert

Ich hoffe, daß Sie das erste C-Programm zum Laufen gebracht haben. Das folgende Programm ist nur eine Umformulierung des ersten Programms, verwendet aber Sprachmittel, die für C typischer sind als die im ersten Beispiel verwendeten Sprachmittel. Dazu gehören der Einsatz von **Funktionen** und der Einsatz von **Adreßvariablen** (Pointern). Das Programm schöpft den Rahmen dessen aus, was ohne systematische Erklärung und ohne Theorie verständlich ist. Wenn Sie gleich an kompliziertere Programme gehen, sollten Sie also die folgenden Kapitel zuvor lesen.

```
/****************************************

     Erstes sehr einfaches C-Programm
          last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Das Programm zaehlt die Blanks in einem Satz
Modifizierte Version mit zwei Prozeduren

****************************************/

#include <stdio.h>

#define  ENDLOS  1

/* Einlesen eines Satzes                    */
/* Liefert die Anfangsadresse eines Satzes */
char *lesen()
{
  static char satz[80];

  printf("\nBitte einen Satz eingeben:");
  gets(satz);
  return(satz);
}   /* end lesen */

/* Liefert die Anzahl Blanks in einem Satz */
int zaehlen(satz)
char *satz;
{
  int anzahl;
```

```
  anzahl = 0;
  while (*satz != 0)
    if (*satz++ == ' ') anzahl += 1;
  return(anzahl);
}   /* end zaehlen */

main()
{
  int anzbl,n;
  char *satzadr;

  while (ENDLOS)
    {
     satzadr = lesen();
     if (*satzadr == 0) break;
     printf("\nDer Satz hat %d Blanks",zaehlen(satzadr));
    }  /* while ENDLOS */
  printf("\n%s\n","--- Programmende ---");
}   /* end main */
```

Das Programm tut das gleiche wie das erste Beispielprogramm. Es folgt dem klassischen Programmaufbau Lesen – Auswerten – Ausgeben. Wenn Sie sich das Hauptprogramm *main()* ansehen, so sehen Sie diese Gliederung: zum Einlesen der Daten wird die Prozedur *lesen()* verwendet, zum Auswerten die Prozedur *zaehlen*() und zum Ausgeben die Prozedur *printf(),* die eine Prozedur der C-Bibliothek ist und deshalb hier nicht im Quellcode erscheint.

Gegenüber der ersten Version ist diese Version (so hoffe ich) übersichtlicher: die Verarbeitungsschritte sind aus *main()* besser erkennbar, weil *main()* kürzer ist als in der ersten Version. Typisch für C-Programme ist in der Tat die Aufteilung eines Gesamtprogramms in kurze, übersichtliche Prozeduren.

Vielleicht sind Sie verwirrt durch die niedergeschriebene Reihenfolge der Prozeduraufrufe: der Aufruf von *zaehlen()* steht nach dem Aufruf von *printf()* und erscheint zudem in der Argumentliste von *printf().* Das muß nicht so sein. Ich hätte das fragliche Programmstück auch so formulieren können:

```
nblanks = zaehlen(satzadr);
printf("\nDer Satz hat %d Blanks",nblanks);
```

Diese Variante stimmt nun in der Schreibreihenfolge mit der Verarbeitungsreihenfolge überein. Erforderlich ist freilich eine zusätzliche Deklaration von *nblanks.*

Meine gewählte Variante – Aufruf von *zaehlen()* im Argument von *printf()* – weist darauf hin, daß in C (und in vielen anderen höheren Programmiersprachen) die physische Reihenfolge der einzelnen Befehle nicht so entscheidend ist wie die Abarbeitungsreihenfolge. Diese aber erfolgt zeitlich so: Auswertung der Argumente, dann Aufruf der Prozedur. Was heißt Auswertung der Argumente? In unserem Fall heißt das, daß nicht die Prozedur *zaehlen()* als Argument übergeben wird, sondern der Wert, der durch diese Prozedur ermittelt wurde. In C liefern alle Prozeduren Werte.

gramm weist zwei solche Funktionen auf: *lesen()* liefert die Anfangsadresse des gelesenen Satzes und *zaehlen()* die Zahl der Blanks im Satz.

Der Typ von Funktionen ist der Typ, der als Funktionswert geliefert werden soll. Er muß in der Prozedurdeklaration angegeben werden. *zaehlen()* liefert einen Wert vom Typ *int, lesen()* einen Wert vom Typ *char* *, also einen Zeiger (Pointer) auf eine Variable vom Typ *char*. Das Sternchen * weist in C (außer in Multiplikationen) auf den Einsatz eines Pointers hin. Der Funktionswert wird im *return*-Befehl übergeben. In der Prozedur *zaehlen()* wird eine Adreßvariable namens *satz* vom Typ *char* * als Argument deklariert. In der Verwendung des Arguments *satz* erscheint auch ein Stern; diesmal ist es der Sternoperator. Der Ausdruck **satz* ist so zu interpretieren: 'liefere das Zeichen, das unter der in *satz* enthaltenen Adresse zu erreichen ist'. Der Ausdruck **satz* verhält sich also wie eine normale Variable vom Typ *char*. Der Ausdruck **satz++* weist sogar zwei Operatoren auf: den Sternoperator und den Inkrementoperator. Dieser Ausdruck läßt sich so umschreiben: 'liefere das Zeichen, das unter der in *satz* enthaltenen Adresse zu erreichen ist und erhöhe anschließend diese Adresse'. Dieser Ausdruck könnte in die beiden folgenden Befehle zerlegt werden:

```
if (*satz == ' ') anzahl += 1;
satz++;  /* gleich satz = satz + 1; */
```

Funktionsnamen sind keine Variablen, sondern Adreßkonstanten. Sie können also nicht wie in Fortran oder Pascal einen Wert an einen Funktionsnamen zuweisen.

Die Prozedur *lesen()* weist, obwohl sie doch sehr einfach ist, zwei Merkwürdigkeiten auf: der Array *satz* ist als *static* deklariert und im *return*-Befehl erscheint ein Arrayname, obwohl die Prozedur doch einen Datentyp *char* * liefern soll. Die letzte Merkwürdigkeit ist sehr einfach zu erklären: Arraynamen sind in C Pointerkonstanten (= Adressen im Speicher). Deshalb ist der Befehl *return(satz);* völlig legal. In C gibt es eine sehr enge Verbindung zwischen Arrays und Pointern. Die Prozedur zeigt auch, daß Sie in C mit der Einschränkung leben können, wonach nur einfache Datentypen (Arrays sind zusammengesetzte Datentypen) als Funktionswert geliefert werden können. Statt eines kompletten Arrays wird ein Zeiger auf das Anfangselement – eben der Arrayname – geliefert.

Die zweite Merkwürdigkeit – *static* in der Deklaration eines lokalen Arrays – ist etwas schwieriger zu erkären. Normalerweise werden lokale Variablen, Arrays und Strukturen auf dem Stack abgelegt. Auf dem Stack werden auch Prozedurargumente und die Rückkehradresse für die Prozedur abgelegt. Meine Prozedur *lesen()* liefert aber eine Adresse, die ohne die *static*-Deklaration im Stack zu suchen wäre. Wenn nun zwischen dem *return*-Befehl und der Auswertung des Rückgabewertes von *lesen()* weitere Prozeduraufrufe erscheinen, wird der Stack überschrieben, so daß mein Rückgabewert zerstört wird. Die *static*-Deklaration sorgt dafür, daß der lokale Array *satz* nicht auf dem Stack, sondern im normalen Variablenbereich abgelegt wird.

Damit ist sichergestellt, daß die in *satz* abgelegten Daten vor Prozeduraufrufen geschützt sind.

Warum Prozedurargumente und Prozeduren? Prozeduren machen Programme vielfältiger und übersichtlicher. Während z.B. in der allerersten Version des Programms nur die Blanks im Array *satz* gezählt werden können, kann ich an die Prozedur *zaehlen()* beliebige Argumente übergeben. Mein Argument kann *satzadr, satz, phrase, sentence* oder irgendwie heißen. Die Prozedur *zaehlen()* verwendet statt des aktuellen Arguments (so heißt das im Aufruf tatsächlich eingesetzte Argument) ein formales Argument (hier heißt es *satz*). Die Prozedur ist also auf viele ähnlich gelagerte Probleme anwendbar.

4 Daten und Datentypen

4.1 Daten und Befehle

Programme sind im Normalfall Sammlungen von Vorschriften, wie aus Daten neue Daten gewonnen werden können. Die Vorschriften werden durch Befehle formuliert; die Daten sind entweder Konstanten oder Variablen.

Wenn Sie sich Ihr Compilerhandbuch oder das Linkerhandbuch vornehmen, werden Sie herausfinden, daß Daten und Befehle in unterschiedlichen Speicherbereichen abgelegt werden; diese unterschiedlichen Speicherbereiche werden auch Segmente genannt. Die Datensegmente werden oftmals weiter unterteilt in Segmente für Konstanten, für initialisierte Variablen und für nichtinitialisierte Variablen. Wozu diese Aufteilung in Segmente gut ist? Sie stellt – um nur einen Grund zu nennen – einen gewissen Schutzmechanismus dar. Befehle und Konstanten dürfen nicht überschrieben werden. Es ist leichter, einen großen Bereich (ein Segment) als einzelne, womöglich wild verstreute kleine Speicherbereiche zu überwachen. Für Spezialanwendungen, wie die Erstellung von ROM-residenten Programmen, ist diese Aufteilung in Segmente unerläßlich. Die Aufteilung in Daten und Befehle ist also intuitiv und compilertechnisch verständlich. Nahezu alle höheren Programmiersprachen (Lisp ist wie fast immer eine Ausnahme) machen diese Unterteilung.

In C erkennen Sie Befehle am Semikolon, das Befehle abschließt. Daten sind entweder als Konstanten (auch sie haben eine feste Syntax) oder als Variablen abgelegt, die eigens deklariert werden müssen. Befehle haben sehr oft ein Schlüsselwort, das sie als Befehl kennzeichnet. Daneben gibt es aber auch Ausdrücke (Kombinationen von Konstanten, Variablen und Operatoren), die selbständig stehen können und den Status von Befehlen haben, wenn sie mit einem Semikolon abgeschlossen sind. Das folgende kurze Beispielprogramm zeigt Konstanten, Variablen und Befehle. Die mit #

beginnenden Zeilen sind Befehle für den C-Präprozessor und zählen nicht zu den C-Sprachelementen im engeren Sinne.

```
#include <stdio.h>
#include <math.h>

main()
{
float wurzel;
  wurzel = sqrt(2.0);
  printf("\nWurzel aus 2 = %f",wurzel);
}   /* end main */
```

Das Programm zeigt drei Befehle (drei Semikolons), eine Variable *wurzel* und zwei Konstanten (2.0 und den String in *printf()*). Die Variablendeklaration *float wurzel;* zählt ebenfalls als Befehl.

4.2 Konstanten und Variablen

Ich habe bereits erwähnt, daß Konstanten und Variablen meist in unterschiedlichen Speicherbereichen abgelegt werden. Es gibt also eine technische Trennung innerhalb der Daten in Konstanten und Variablen. Es gibt aber auch eine C-interne Begründung, die eine solche Trennung erkärt.

In C sind Konstanten Datenobjekte, die nur vom Wert und vom Datentyp, nicht aber vom Speicherplatz her interessant sind. Konstanten haben keine Speicheradresse im C-Sinne! Ganz konsequent ist das allerdings nicht: Stringkonstanten bilden die Ausnahme. Zudem haben Konstanten keinen Namen – sie bezeichnen sich selbst. Variablen können unter mindestens vier Aspekten betrachtet werden:

- sie haben einen Namen
- sie haben einen Wert
- sie haben einen Datentyp
- sie haben eine Adresse im Speicher

In der Deklaration einer Variablen – jede Variable sollte deklariert werden – wird der Name der Variablen festgelegt und eventuell weitere Attribute wie der Gültigkeitsbereich, die Speicherklasse und vielleicht der Wert selbst, wenn die Variable schon beim Deklarieren initialisiert wird. Die Adresse im Speicher kann vom Programmierer nicht festgelegt werden – dafür sorgt schon der Compiler und später der Linker.

Die Unterscheidung in Konstanten und Variablen hat zwei wichtige Konsequenzen in C-Programmen: auf Konstanten kann der Adreßoperator & nicht angewandt werden

(er ermittelt die Adresse einer Variablen) und Konstanten können nicht auf der linken Seite von Zuweisungen vorkommen. Der Speicherbereich einer Konstanten – falls es einen solchen gibt – ist für C-Programmierer weitgehend tabu.

4.3 Datentypen

4.3.1 Wozu Datentypen?

Variablen und Konstanten haben in C zwei gemeinsame Aspekte: sie haben einen Wert und einen Datentyp. Die Konstante 17 etwa hat den Wert 17 und den Datentyp *int* – sie stellt eine ganze, vorzeichenbehaftete Zahl dar. Die Konstante 17.0 dagegen hat zwar den gleichen Wert, aber den Datentyp *float*; sie stellt eine Fließkommazahl dar. Der Datentyp von Konstanten ist an der Schreibweise erkennbar. Fließkommakonstanten sind z.B. am Punkt zu erkennen. Der Datentyp von Variablen ist nicht aus dem Variablennamen erkennbar. Deshalb sollte der Datentyp in den Deklarationen von Variablen immer angegeben werden – sonst wird *int* als Datentyp angenommen. Es gibt keine Regel wie in Fortran, wonach die Anfangsbuchstaben den Datentyp festlegen, wenn nichts anderes genannt ist.

Der Datentyp einer Variablen oder einer Konstanten legt zweierlei fest:

- die Größe des reservierten Speicherplatzes (bei Variablen)
- die Operationen, die auf das Objekt anwendbar sind

Je nach Datentyp werden z.B. in C für ganze Zahlen 1 Byte (Typ *char*), 2 Bytes (Typen *short* und *int*) oder vier Bytes (Typ *long*) reserviert, für Fließkommazahlen 4 Bytes (*float*) oder 8 Bytes (*double*). Der Datentyp legt also die Größe des reservierten Speichers fest.

Daneben wird aber auch festgelegt, ob – um beim Beispiel der ganzen Zahlen zu bleiben – normale Arithmetik oder vorzeichenlose Arithmetik eingesetzt werden kann. Der Datentyp *unsigned* legt fest, daß auf ein Objekt vorzeichenlose Arithmetik angewandt werden soll. Beispiel:

```
/****************************************

              Arithmetik-Test
            last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Testet signed und unsigned Arithmetik

****************************************/
```

```
#include <stdio.h>

main()
{
  unsigned uvar;
  int      svar;

  svar = 32767;
  uvar = 32767;
  svar = svar + 2;
  uvar = uvar + 2;
  printf("\nsvar = %d uvar = %u",svar,uvar);
}   /* end main */
```

Das Programm liefert die Ergebnisse *svar = -32767* und *uvar = 32769* (für *uvar* und *svar* wurden je zwei Bytes reserviert). Bei der Variablen *svar* findet ein Überlauf in den negativen Bereich statt, weil *32767* (hex *7fff*) schon die größte positive Zahl für den Datentyp *int* auf einem IBM-PC darstellt. Für den Datentyp *unsigned* liegt dagegen die größte Zahl erst bei *65535* (hex *ffff*). Die Bitmuster der Ergebnisse sind beide Male gleich; die Interpretation der Ergebnisse ist allerdings verschieden.

Wenn Sie auf dem Taschenrechner die Zahl *17* durch die Zahl *18* dividieren, erhalten Sie das Ergebnis *0.9444...* Das gleiche Ergebnis liefert das folgende Codestück:

```
/* float - Division */
float z1,z2;
  z1 = 17.0;
  z2 = 18.0;
  printf("\nErgebnis = %f", z1 / z2);
```

Wenn Sie aber auf die Idee kommen, die Variablen *z1* und *z2* als ganze Zahlen zu deklarieren, wird Sie das Ergebnis vielleicht verblüffen:

```
/* Integer - Division */
int z1,z2;
  z1 = 17;
  z2 = 18;
  printf("\nErgebnis = %d", z1 / z2);
```

Dieses Programmstück liefert das Ergebnis 0, weil jetzt Integer-Arithmetik statt Fließkomma-Arithmetik getrieben wird. Es hilft auch nichts, wenn im Ergebnisausdruck *%f* statt *%d* eingesetzt wird, weil dann eine *float*-Zahl als Argument von *printf* erwartet wird, die jedoch nicht übergeben wurde. Der Datentyp der Operanden des Divisionsoperators entscheidet über die Rechenoperationen, die zur Ermittlung des Ergebnisses eingesetzt werden.

4.3.2 Hierarchien von Datentypen

In C kann man vier Stufen von Datentypen unterscheiden:

- Basistypen (einfache Typen): *char, unsigned char, short, unsigned short, int, unsigned, long, unsigned long, float, double*
- abgeleitete Datentypen: Funktionen und Pointer (Maschinenadressen)
- zusammengesetzte Datentypen: Arrays, Strukturen und Verbunde
- benutzereigene Datentypen: sie werden über *typedef* definiert

In der Originalversion von K&R gibt es übrigens die Typen *unsigned char, unsigned short* und *unsigned long* nicht! Der Typ *unsigned* ist mit *unsigned int* gleichbedeutend. Die Fließkommatypen *float* und *double* sind immer vorzeichenbehaftet. Aus den Basistypen können die abgeleiteten und die zusammengesetzten Datentypen aufgebaut werden. Zur Unterscheidung von den Basistypen dienen einige Schreibkonventionen:

- Funktionen: runde Klammern hinter dem Funktionsnamen oder dem Funktionspointer.
- Pointer: Stern in der Deklaration.
- Array: eckige Klammern in der Deklaration und im Zugriff auf Arrayelemente.
- Strukturen: Schlüsselwort *struct* in der Deklaration, Punktoperator oder Pfeiloperator (s. Kap. 5) im Zugriff.
- Verbunde: Schlüsselwort *union* in der Deklaration, Punktoperator oder Pfeiloperator (s. Kap. 5) im Zugriff.

In C gibt es keine Vorschrift, wie groß die Datenobjekte sein müssen, die Basisdatentypen als Attribut haben. *int*-Zahlen können z.B. 2 Bytes (8086-PCs, 68000 PCs) oder vier Bytes (VAX) groß sein. Die Größe der Datentypen wurde deswegen nicht festgelegt, weil nicht auf allen Maschinen alle Größen gleich effizient berechnet werden können. Die Größe einer *int*-Zahl sollte die 'natürlichste' Datengröße Ihrer Maschine sein, bei einer 16-Bit-Maschine also 16 Bit, bei einer 32-Bit-Maschine 32 Bit etc. Es ist also keine Schwäche von C, wenn die Datengrößen nicht eindeutig festgelegt sind, sondern Rücksicht auf Recheneffizienz.

Ab und zu (z.B. wenn Sie abschätzen wollen, wie genau Sie rechnen können) ist es wichtig, die Größe von Variablen in Bytes zu kennen. Dazu können Sie den *sizeof*-Operator benutzen. Bitte testen Sie das nachfolgende Programm an Ihrer Maschine. Achten Sie bitte auch darauf, bei welchen Datentypen Ihr Compiler eventuell Fehler meldet:

```
/****************************************

            Testet Datentypen
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Test auf Groesse und Vorhandensein von Einfachen und abgeleiteten
Datentypen

****************************************/

#include <stdio.h>

main()
{
  /* Basisdatentypen */
  printf("\nGroesse von char:           %d",sizeof(char));
  printf("\nGroesse von unsigned char:  %d",sizeof(unsigned char));
  printf("\nGroesse von short:          %d",sizeof(short));
  printf("\nGroesse von unsigned short: %d",sizeof(unsigned short));
  printf("\nGroesse von unsigned:       %d",sizeof(unsigned));
  printf("\nGroesse von long:           %d",sizeof(long));
  printf("\nGroesse von unsigned long:  %d",sizeof(unsigned long));
  printf("\nGroesse von float:          %d",sizeof(float));
  printf("\nGroesse von double:         %d",sizeof(double));
  /* einige abgeleitete Datentypen */
  printf("\nGroesse von (int *):        %d",sizeof(int *));
  printf("\nGroesse von (char *):       %d",sizeof(char *));
}   /* end main */
```

Meine Ergebnisse finden Sie in der nachfolgenden Tabelle.

	MSC (small model)	**MSC (big model)**	**VAX**	**Amiga**
Groesse von char:	1	1	1	1
Groesse von unsigned char:	1	1	1	1
Groesse von short	2	2	2	2
Groesse von unsigned short:	2	2	2	2
Groesse von int:	2	2	4	2
Groesse von unsigned:	2	2	4	2
Groesse von long:	4	4	4	4
Groesse von unsigned long:	4	4	4	4
Groesse von float:	4	4	4	4
Groesse von double:	8	8	8	8
Groesse von (int *):	2	4	4	4
Groesse von (char *):	2	4	4	4

4.3.3 Eigenschaften der Basistypen

Zwei Klassen von Zahlen sind in den Basistypen enthalten: Ganze Zahlen (Integer) und Fließkommazahlen. Der Begriff 'Fließkommazahl' ist hier übrigens etwas irreführend, da anstelle des Kommas ein Punkt den ganzzahligen vom gebrochenen Anteil trennt. In der internen Darstellung einer Fließkommazahl erscheint weder Punkt noch Komma.

Die Klasse der **ganzen Zahlen** wird durch die Basistypen *char, short, int* und *long* vertreten; diese Typen repräsentieren die **vorzeichenbehaftete** Variante der ganzen Zahlen. Daneben gibt es **vorzeichenlose** Varianten, von denen mindestens der Typ *unsigned* in allen C-Compilern vertreten ist. Während bei den vorzeichenlosen Typen alle Bits des reservierten Speicherbereichs zur Zahlendarstellung herangezogen werden, wird bei den vorzeichenbehafteten Typen das vorderste Bit (MSB = most significant bit) für die Darstellung des Vorzeichens benutzt. Hier die Tabelle der Zahlenbereiche für den Amiga:

Typ	Bereich von	bis
char	-128	+127
unsigned char	0	+255
int (short)	-32768	+32767
unsigned	0	+65535
long	-2147483648	+2147483647
unsigned long	0	+4294967295

Der Zusammenhang zwischen vorzeichenlosen und vorzeichenbehafteten Zahlen soll an einem Beispiel demonstriert werden. Nehmen wir an, Sie wollen wissen, welches Bitmuster der Zahl -1 entspricht. Sie schreiben +1 als Bitmuster auf:

```
0000 0000 0000 0001    = 0x0001
```

Dann invertieren Sie alle Bits dieses Musters (Sie bilden das Komplement):

```
1111 1111 1111 1110    = 0xfffe
```

Und im letzten Schritt addieren Sie eine 1 und erhalten das Zweier-Komplement:

```
1111 1111 1111 1111    = 0xffff
```

Der Typ *char* kann – trotz seines Namens – für ganz normale Zahlenarithmetik verwendet werden; meist wird er jedoch zur Aufnahme von Zeichen in 8-Bit-Darstellung verwendet. Die vorzeichenlosen Typen (z.B. *unsigned*) werden besonders gern beim Zugriff auf Arrays eingesetzt, da Arrays in C üblicherweise beim Element 0 beginnen und negative Indexwerte kaum Sinn machen. Mit einer Indexvariablen vom Typ *unsigned* können dann immerhin doppelt soviele Elemente angesprochen werden wie mit einer Indexvariablen vom Typ *int*.

4.3.4 Konvertierungsregeln

Es gibt Fälle, in denen Daten automatisch vom Compiler konvertiert werden, d.h. sie wechseln den Datentyp. Solche Fälle liegen vor, wenn

- zwei oder mehr Datenobjekte von verschiedenem Datentyp in einem Ausdruck (z.B. einer Formel, einer Zuweisung) vorkommen.
- Datenobjekte als Argumente an eine Prozedur übergeben werden.

Typische Beispiele dafür sind:

```
/* Beispiele fuer Typkonvertierung */
/* Deklarationen mit Initialisierung */
int ivar   = 10;
long lvar  = 5L;
short svar = 2;
char cvar  = '\n';
float fvar = 2.0;

    /* 1.verschiedene Typen in einem Ausdruck      */
    /* ivar ---> long                              */
    if ((lvar + svar) > ivar) lvar = ivar;

    /* 2.Uebergabe an eine Prozedur                */
    /* cvar ---> int                               */
        if (maximum(ivar,cvar) == ivar) cvar = 0xff;

        /* 3. ditto, float nach double, int nach float */
        if (fmaximum(fvar + 1, 2.0) == 2.0)  fvar = 2.0;
```

Die Konvertierungsregeln, nach denen die C-Compiler vorgehen, sind grob gesehen die folgenden:

- Jedes Objekt vom Typ *float* wird in Berechnungen und bei der Argumentübergabe nach *double* konvertiert (Beispiel 3)
- Jedes Objekt in Berechnungen oder bei der Argumentübergabe, das kürzer ist als *int*, wird nach *int* konvertiert (*cvar* in Beispiel 2)
- *unsigned* Arithmetik hat in Ausdrücken Vorrang vor *signed* Arithmetik
- Der Typ des längsten Objekts hat Vorrang (*lvar*, Datentyp *long* in Beispiel 1)
- Fließkommaarithmetik hat Vorrang vor ganzzahliger Arithmetik.

Von diesen Regeln sollen zwei nochmals in leicht anderer Formulierung wiederholt werden, weil sie besonders wichtig sind: *float*-Objekte werden vom Lattice-C-Compiler in Berechnungen und bei der Argumentübergabe immer nach *double* konvertiert, und: kein übergebenes Argument kann kürzer als *int* sein. Damit sind die folgenden Übergaben der aktuellen an die formalen Argumente korrekt. Beachten Sie die unterschiedlichen Deklarationen von *cvar* und *iarg*:

```
/* Erstes Beispiel      */
/* akt. Argument = char */
char cvar;
    funct1(cvar);
    ......
```

```
/* formales Argument = int */
funct1(iarg)
int iarg;
{
...
}   /* end funct1 */

/* Zweites Beispiel       */
/* akt. Argument = float */
float fvar;
  funct2(fvar);
  ........
/* formales Argument = double */
funct2(darg)
double darg;
{
  .....
}   /* end funct2 */
```

Bei der Konvertierung kürzerer ganzzahliger Objekte in längere ganzzahlige Objekte wird das Vorzeichenbit in die freien Bits kopiert. Man nennt diesen Vorgang 'sign extension". Ich erwähne diese Regel besonders, weil sie oftmals (auch von mir) vergessen wird. Besonders heimtückisch wirkt sie sich aus beim Abfragen der deutschen Umlaute, die meist in *ASCII*-Positionen > *0x80* angesiedelt werden, also das vorderste Bit in *char*-Variablen gesetzt haben. In diesem Fall darf nicht – wie folgt – das 'ä' des Amiga als Dezimalwert 132 abgefragt werden, sondern als -124:

```
/* falsche Version: sign extension! */
char c;
  c = 'ä';
  if (c == 132) ....
/* drei korrekte Versionen */
unsigned char c;
c = 'ä';
  if (c == 132) ....
char c;
  c = 'ä';
  if ((c & 0xff) == 132) ...
  if (c == -124) ...
```

Wenn Sie in Formeln, in Zuweisungen oder bei der Argumentübergabe einen bestimmten Typ erzwingen wollen, können Sie den *Cast-Operator* (Typerzwingungsoperator) verwenden. Dazu müssen Sie den gewünschten Datentyp in runde Klammern vor das Objekt setzen, den gewünschten Typ erhalten soll. Beispiele:

```
/* statt : xyz = sqrt(2.0); */
xyz = sqrt((float)2);
/* IO - Byte (Adresse 3) in CP/M setzen */
*((unsigned char *)3) = 0x84;
```

4.4 Schreibregeln

4.4.1 Schreibregeln für Konstanten

Jede Programmiersprache – so auch C – hat eigene Schreibkonventionen für Konstanten. Es gibt in C definierte Schreibregeln für Zahlen in verschiedenen Datentypen, für Zeichenkonstanten und für Strings. Strings sind eigentlich kein offizieller Datentyp in C. Sie werden jedoch in den verschiedenen C-Implementierungen einheitlich behandelt, so daß sie hier nicht vernachlässigt werden dürfen. Es fällt Ihnen vielleicht auf, daß die Booleschen Konstanten 'Wahr' und 'Falsch' nicht erscheinen. Der Grund ist ganz einfach: es gibt in C keinen Datentyp, der dem Typ *Boolean* in Pascal oder dem Typ *Logical* in Fortran entsprechen würde. Der ganzzahlige Wert 0 wird wie 'Falsch' behandelt, alle anderen ganzzahligen Werte wie 'Wahr'.

Beginnen wir mit den ganzen Zahlen. Falls Sie im **Dezimalsystem** bleiben, brauchen Sie nichts zu lernen: die Zahlen werden wie in der Schule notiert, also z.B. '17', oder '-1'. Die Notation '+17' statt '17' wird übrigens nicht akzeptiert! Der C-Compiler nimmt den Typ *int* für diese Konstanten an, also den Typ 'ganze vorzeichenbehaftete Zahl', wenn der Wertebereich nicht überschritten wird.

Konstanten vom Typ *long* können Sie mit einem nachgestellten 'l' oder 'L' kennzeichnen. Wenn der Wertebereich den für *int* definierten Bereich überschreitet, wird sowieso *long* als Datentyp angenommen. Ein '+' als Vorzeichen für positive Zahlen ist wieder nicht erlaubt. Sie können auch Konstanten mit dem Typerzwingungsoperator (cast-Operator) versehen wie im nachfolgenden Beispiel:

```
/* verschiedene Varianten fuer long - Zahlen */
long la;
  la = 40000l;
  la = 40000L;
  la = -40000L;
  la = -4l;
  /* mit cast Operator */
  la = (long)40000;
```

Der Ziffernbereich von Dezimalkonstanten liegt im Bereich '0' - '9'. **Hexadezimalkonstanten** haben die Zeichenkombination '0x' als Präfix und bestehen aus Kombinationen der Zeichen '0' - '9' und 'A' - 'F' oder 'a' - 'f'. Die Zahl 139 wird hexadezimal als *0x8b* notiert. Vorlaufende Nullen dürfen ebenfalls erscheinen: Sie können also

auch *0x008b* schreiben. Zur Erinnerung: den Dezimalwert dieser Zahl können Sie berechnen als

```
(8 * (16 hoch 1)) + (b * (16 hoch Ø))
```

Die Hexadezimalziffer b hat den Wert 11. Also rechnet sich 0x8b als

```
(8 * (16 hoch 1)) + (11 * (16 hoch Ø))
```

was 139 dezimal ergibt. Hexadezimalzahlen gruppieren immer vier Bits zu einer Ziffer. Ein Byte (8 Bits) läßt sich also mit zwei Hex-Ziffern darstellen, eine *int*-Konstante mit vier Ziffern (vorausgesetzt, *ints* (Integer-Werte) werden in zwei Bytes dargestellt). Standard-C läßt die Notation von Binärkonstanten nicht zu. Deshalb hat sich die Hexadezimalnotation als Alternative für die Notation von Binärkonstanten durchgesetzt. Sie kommen also nicht umhin, sich diese Schreibweise anzueignen.

Oktalzahlen beginnen mit einer vorlaufenden Null. (Für Oktalzahlen in Strings und im Input gelten leider etwas andere Regeln.) Der zulässige Ziffernbereich geht von '0' - '7'. Die Dezimalzahl 139 wird oktal 0213 notiert. Auch hier dürfen Sie weitere Nullen (nach der ersten, obligatorischen Null) verwenden. Rechnen wir die Oktalzahl 0213 dezimal um: 0213 =

```
(2 * (8 hoch 2)) + (1 * (8 hoch 1)) + (3 * (8 hoch Ø))
```

Das Oktalsystem ist bei den heutigen Computern nicht mehr sehr verbreitet, weil die Wortbreiten des Speichers meist Vielfache von 8 sind und andererseits das Oktalsystem immer drei Bits zu einer Ziffer zusammenfaßt. In C hat sich allerdings das Oktalsystem als Relikt aus der 8-Bit-Rechner-Zeit gehalten.

In Strings lassen sich nicht darstellbare ASCII-Zeichen als Oktalzahlen mit vorgestelltem Backslash notieren. Hexadezimale oder dezimale Darstellung ist dort nicht möglich. Wenn Sie also C komplett beherrschen wollen, sollten Sie die Oktalschreibweise der wichtigsten ASCII-Kontrollcodes und des Bereichs von 0x80 - 0xff (erweiterter ASCII-Bereich) kennen.

Auch den Oktal- und Hexadezimalkonstanten können Sie ein 'l' oder 'L' anhängen, um so lange Zahlen zu kennzeichnen.

Für **Fließkommakonstanten** gibt es in C zwei Notationen: die normale Notation (z.B. 3.414) und die wissenschaftliche Notation (z.B. *718.23E7* oder *718.23e7*). Beide Notationen weisen den Punkt als Trenner zwischen ganzzahligem und gebrochenem Anteil auf. Ein 'E' oder 'e' trennt in der wissenschaftlichen Notation die Mantisse vom Exponenten. *718.23E7* wird interpretiert als *718.23 * (10 hoch 7)*. Fließkommazahlen können wie ganze Zahlen mit einem negativen Vorzeichen versehen werden. Fließkommakonstanten werden immer als *double*-Werte betrachtet.

Zeichenkonstanten werden zwischen einfachen Hochkommata (Apostrophen) notiert. Dabei sind drei Schreibvarianten zu unterscheiden:

- Zwischen den Hochkommata erscheint ein einfaches, darstellbares Zeichen, z.B. *'a'* oder *'Z'* oder *'.'*.
- Zwischen den Hochkommata erscheint ein Backslash gefolgt von einem Kleinbuchstaben aus einer festen Liste (s. unten).
- Zwischen den Hochkommata erscheint ein Backslash gefolgt von einer bis zu dreistelligen Oktalzahl.

Die beiden letzten Schreibvarianten (Ersatzdarstellungen) werden vor allem zur Darstellung nicht darstellbarer Zeichen verwendet. So kann z.B. das Zeichen Linefeed (LF) als *'\n'* oder '\012' notiert werden. Von den Compilern werden die folgenden Kurznotationen erkannt:

```
\n    Linefeed
\r    Carriage return
\b    Backspace
\t    Tabulator
\g    Bell
```

Da die Zeichen \(Backslash), einfaches Hochkomma und doppeltes Hochkomma einen Sonderstatus einnehmen, müssen sie in Zeichenkonstanten so notiert werden:

```
\\     Backslash
\'     einfaches Hochkomma
\"     doppeltes Hochkomma
```

Strings werden in C als Kette von Zeichenkonstanten zwischen doppelten Hochkommata notiert. Der Ausdruck *"Das ist ein String"* ist also eine Stringkonstante. Stringkonstanten sind die einzigen Konstantenausdrücke in C, deren Adresse Sie ermitteln können. Für Strings gilt eine besondere Ablage: sie erhalten eine binäre Null als Endekennung. Ein String wie "ABC" kann als abkürzende Notation von

```
'A', 'B', 'C', '\0'
```

verstanden werden. In Strings ist die erwähnte Ersatzdarstellung für Zeichenkonstanten erlaubt wie im folgenden Beispiel innerhalb eines *printf()*-Aufrufs:

```
printf("\n\t\tCopyright E.Huckert\n");
/* identisch mit dem folgenden Ausdruck */
printf("\012\011\011Copyright E.Huckert\012");
```

Ein sehr wichtiger Hinweis: Strings werden vom Compiler in besonderen Datenbereichen abgelegt. Zur Compilezeit werden Strings im Code durch die Anfangsadresse ersetzt, an der der String abgelegt wurde. (Strings in Arrayinitialisierungen folgen

nicht dieser Regel). An *printf()* wird also im obigen Beispiel nicht die dort stehende Zeichenkette selbst übergeben, sondern nur eine Adresse darauf. Strings können in C – ähnlich wie Arrays – als Pointerkonstanten betrachtet werden. Deshalb also sind Programme wie das folgende zulässig:

```
/* Strings sind Pointerkonstanten */
myprint(sadr)
char *sadr;
{
  while (*sadr != 0)
    putchar(*sadr++);
}

main()
{
   myprint("\nEs wird was ausgegeben");
}
```

Wie Sie sehen, hat *myprint()* eine Stringkonstante als aktuelles Argument, weist aber als formales Argument einen Pointer auf.

Es kann zu Komplikationen kommen, wenn Sie versuchen, einen String zur Laufzeit zu überschreiben, da Strings bisweilen in schreibgeschützten Bereichen abgelegt werden. Vorsicht also bei Programmstücken wie dem folgenden. Sie sind nicht transportabel:

```
/* Stringkopie in Konstante: Vorsicht!!! */
char *hadr;
    hadr = "ABCD";
    strcpy(hadr,"abcd");
```

4.4.2 Schreibregeln für Variablen und andere C-Objekte

Die Schreibregeln für Variablen – damit sind insbesondere die Regeln für die Wahl eines Variablennamens gemeint – entsprechen den Regeln, die auch für andere C-Objekte wie Funktionsnamen, Namen von Verbunden, Arraynamen, Strukturnamen, Sprungmarken oder *typedef*-Namen gelten.

Die wichtigste Regel: in C werden Klein- und Großbuchstaben nicht gleich behandelt. Eine Variable namens *xyz* ist also verschieden von einer Variablen *XYZ*. Bitte beachten Sie das; die meisten Betriebssysteme (Ausnahme UNIX) behandeln Ausdrücke in Groß- und Kleinschreibung gleich.

Nach K&R können die Namen von C-Objekten nur mit Buchstaben oder dem Zeichen '_' beginnen. Dann dürfen beliebige Kombinationen aus Zahlen, Buchstaben

und dem Zeichen '_' folgen. In einigen Compilern sind weitere Zeichen zugelassen, z.B. das Dollarzeichen. Nach K&R sind nur acht Stellen eines Symbols signifikant; moderne Compiler lassen jedoch mehr Zeichen zu. Falls Ihr Compiler Assemblercode erzeugt, sollten Sie daran denken, daß dann die Schreibregeln des Assemblers Vorrang vor den Schreibregeln des Compilers haben. Meiner Erfahrung nach ist es dann auch gefährlich, Symbole zu verwenden, die den Namen von Maschinenregistern entsprechen.

Die folgenden C-Schlüsselwörter dürfen nicht als Namen von C-Objekten gewählt werden:

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

In neueren C-Compilern, wie auch bei Lattice-C und Atztec-C werden weitere Schlüsselwörter (z.B. *enum* und *void*) verwendet. Meiner Erfahrung nach sollten Sie auch die Namen der Bibliotheksroutinen (s. Kap. 12) nicht für eigene Funktionen oder Variablen verwenden, da diese Namen in manchen *#include*-Files deklariert werden; falls Ihre Deklaration nicht mit einer solchen Deklaration übereinstimmt, entstehen schwer zu lokalisierende Fehler.

Einige Beispiele für zulässige und nicht zulässige Symbole in C:

```
/* Zulaessige Symbole */
int a, _a, A, _A, a_b, A_B, a1, a2;
char symbol1, symbol1a;
extern unsigned _hfunct();

/* gefaehrlich: symbol1ax ist auf acht */
/* Stellen identisch mit symbol1a      */
char symbol1ax;

/* Falsch: Symbol beginnt nicht mit Buchstabe oder _ */
long 1abc;

/* Falsch: Symbol ist mit Schluesselwort identisch */
float double;
```

4.5 typedef

Mit dem Schlüsselwort *typedef* können benutzereigene Datentypen geschaffen werden. Die Schaffung benutzereigener Datentypen steigert die Lesbarkeit von Programmen. Wenn ich z.B. auf einem Amiga ausdrücken will, daß ich Variablen mit 16 Bits und mit 32 Bits verwende, dann kann ich dies so tun:

```
typedef long VAR32;
typedef short VAR16;
```

In meinen Variablendeklarationen, in *sizeof*-Ausdrücken und in Ausdrücken mit dem Cast-Operator kann ich jetzt *VAR32* und *VAR16* genauso wie *long* und *short* verwenden:

```
/* Deklaration von Variablen */
VAR32 xvar1,xvar2;
VAR16 yvar1,yvar2;
/* sizeof - Ausdruck */
pvar = malloc(sizeof(VAR32));
/* cast - Operator */
if ((VAR32)yvar1 > xvar1) yvar1 = xvar1;
```

Das Schlüsselwort *typedef* erzeugt Synonyme, also gleichbedeutende Datentypen. Häufiger noch als im obigen Beispiel für Basistypen wird *typedef* zur Definition von Synonymen für zusammengesetzte Datentypen (insbesondere Strukturen) verwendet.

Das Schlüsselwort *typedef* verhält sich syntaktisch wie die Schlüsselwörter *extern, static* etc. Da in C nur eine Angabe von Gültigkeitsbereichen oder Speicherklassen erlaubt ist, darf *typedef* nicht mit den dafür zuständigen Schlüsselwörtern kombiniert werden.

4.6 Gültigkeitsbereiche und Speicherklassen

Es wurde in Kap. 4.2 schon erwähnt, daß bei der Deklaration von Variablen Gültigkeitsbereiche und Speicherklassen angegeben werden können. Daneben gibt es Regeln, die Gültigkeitsbereiche und Speicherklassen auch ohne explizite Deklaration festlegen. Doch nicht nur Variablen – also Daten – unterliegen Regeln, die den Gültigkeitsbereich festlegen, auch für Prozeduren, Prozedurargumente und Sprungmarken gibt es solche Regeln. Was ist nun ein Gültigkeitsbereich, was eine Speicherklasse?

Der **Gültigkeitsbereich** eines Objekts gibt an, in welchem Bereich (über welche Codestrecke) der Name eines Objekts bekannt ist. Es gibt in C die Gültigkeitsbereiche

- extern
- intern (lokal)

wobei nur das Attribut *extern* als Schlüsselwort existiert. Alles was nicht *extern* ist, ist automatisch *intern*. Im Gegensatz zu Pascal gibt es in C nur externe Prozeduren, da innerhalb von C-Blöcken nicht wieder Prozeduren definiert werden können. Prozeduren können also nicht ineinander geschachtelt sein. Variablen können *extern* oder *intern* sein. Alles, was nicht innerhalb eines Blocks definiert wurde und was nicht Prozedurargument ist, ist extern zu diesem Block. Der Name eines externen Objekts ist innerhalb eines Blocks bekannt, obwohl er nicht im Block deklariert wurde. Objekte, die nach einer öffnenden geschweiften Klammer deklariert werden, sind intern zum jeweiligen Block. Üblicherweise wird das Schlüsselwort *extern* nur dazu benutzt, den Typ von Prozeduren und von Variablen zu deklarieren, die nicht im aktuellen Programm niedergeschrieben sind. Bei externen Variablen, die im gleichen Programm deklariert werden, wird meist auf das Schlüsselwort *extern* verzichtet.

Beispiele für externe und interne Objekte:

```
extern char *finde(); /* nicht im Programm definiert */
char zeile[80];       /* extern zu main */

main()
{
int n;                /* intern zum Block main */
  printf("\nGib drei Saetze ein:");
  for (n=0; n < 3; n++)
  {
    gets(zeile);
    if (finde(zeile,".") == NULL)
    printf("\nAussagesaetze schliesst man mit Punkt ab");
  }
}   /* end main */
```

In diesem Programm kommen zwei externe Objekte vor: der Array *zeile* ist extern zu *main(),* weil er außerhalb von *main()* deklariert wurde; die Prozedur *finde()* wird als *extern* deklariert, weil sie nicht im Programm enthalten ist und außerdem einen Wert liefert, der nicht dem Standardwert *int* entspricht. Die Variable *n* ist innerhalb von *main()* deklariert und deshalb intern. Ihr Name ist nicht außerhalb von *main()* bekannt.

Die Kennwörter *auto, register* und *static* stehen für **Speicherklassen.** Über ein solches Kennwort kann definiert werden, in welcher Art Speicher eine Variable abgelegt werden soll. Die in Frage kommenden Speicherarten sind der Stack, Maschinenregister und der Variablenbereich des normalen Benutzerspeichers.

Variablen mit dem Schlüsselwort *register* sollen in Maschinenregistern (CPU-interne Speicher mit besonders schnellem Zugriff) abgelegt werden. Registervariablen können nur interne Variablen sein. Registervariablen sind ein beliebtes Mittel zur Steigerung der Ablaufgeschwindigkeit von Programmen. Sie haben allerdings eine Einschränkung: auf sie kann der Adreßoperator nicht angewandt werden – Maschinenregister haben keine Adresse, selbst wenn das nicht für alle Maschinen stimmt. Der Compiler ist nicht verpflichtet, das Schlüsselwort *register* auszuwerten. Dieses Schlüsselwort stellt vielmehr eine **Empfehlung** an den Compiler dar. Die Compiler reservieren sich einige Register für interne Zwecke; deshalb kann man nicht unbedingt davon ausgehen, daß für alle Variablen Register eingesetzt werden, die mit diesem Schlüsselwort deklariert wurden.

Variablen mit dem Schlüsselwort *auto* (steht für 'automatic') werden erst beim Eintritt in einen Block auf dem Stack angelegt. Beim Verlassen eines Blocks wird die Variable wieder vergessen – sie ist dann von außen nicht mehr erreichbar. Das Schlüsselwort *auto* wird kaum verwendet, da interne Variablen und Prozedurargumente immer das Attribut *auto* haben, wenn es sich nicht um Registervariablen handelt.

Die formalen Argumente einer Prozedur werden vor der öffnenden Blockklammer der Prozedur deklariert. Sie werden dennoch wie interne Variablen behandelt und werden auch wie solche auf dem Stack angelegt. Auch für Prozedurargumente darf das Schlüsselwort *register* verwendet werden.

Variablen mit dem Schlüsselwort *static* werden nicht auf dem Stack angelegt, sondern im allgemeinen Datensegment; sie sind allerdings nicht außerhalb des aktuellen Programms (bei externen Variablen) oder des aktuellen Blocks (bei internen Variablen) bekannt. *static* kann ebenso wie *extern* auch auf Prozeduren angewandt werden. Prozeduren mit dem Attribut *static* sind nur innerhalb des aktuellen Programms bekannt. Compiler und Linker verstecken deren Namen vor anderen Programmodulen, so daß dort die gleichen Namen verwendet werden können. Lokale *static*-Variablen stehen außerhalb der Rekursion, da sie nicht auf dem Stack angelegt werden. In Kap. 3.5 wurde auch schon gezeigt, wie *static*-Objekte in Funktionen verwendet werden können, die Arrays oder Strukturen als Funktionswerte liefern. Falls Sie mit einem Debugger arbeiten, wird Ihnen auffallen, daß statische Funktionen und statische Variablen vom Compiler 'umgetauft' werden, um Namenskonflikte zu vermeiden.

Sprungmarken dürfen in C nur mit dem *goto*-Befehl verwendet werden. Sie sind nur in der Prozedur bekannt, in der sie erscheinen. Mehrere Prozeduren dürfen also gleichnamige Sprungmarken verwenden, ohne daß es zu Konflikten kommt.

5 Operatoren

5.1 Ausdrücke und Operatoren

Ein C-Programm besteht aus Befehlen. Die Befehle selbst sind aus Ausdrücken und C-Schlüsselwörtern zusammengesetzt. Ein C-Ausdruck ist

- entweder eine Konstante (elementarer Ausdruck)
- oder eine Variable (elementarer Ausdruck)
- oder eine Kombination aus einem Operator und ein oder mehreren Ausdrücken (zusammengesetzter Ausdruck)
- oder ein Ausdruck zwischen runden Klammern

In C können alle Ausdrücke als Befehle verwendet werden, wenn dies sinnvoll ist. Sie werden dann mit einem Semikolon als Befehle gekennzeichnet. Beispiele für Ausdrücke:

```
17
ivar
-17     /* Vorzeichenoperator */
&ivar   /* Adressoperator */
ivar + 18
/* Sternoperator, Gruppierungsklammern */
*(cadr + (ivar * 32))
```

Operatoren sind festdefinierte Elemente der Sprache C, die aus Ausdrücken wieder Ausdrücke machen. Ausdrücke werden zur Laufzeit eines Programms berechnet (evaluiert), d.h. ihr Wert wird ermittelt. Während die Wertermittlung für Konstanten und Variablen einfach ist, kann sie für zusammengesetzte Ausdrücke recht kompli-

ziert sein. Bei der Berechnung eines operatorhaltigen Ausdrucks muß auf mehrere Eigenschaften der Operatoren Rücksicht genommen werden:

- Operatoren haben eine **Stelligkeit**. Es gibt in C ein-, zwei- und dreistellige Operatoren. Die Stelligkeit gibt an, wieviele Ausdrücke der Operator als Operanden hat, d.h. wieviele Ausdrücke durch den Operator zu einem neuen Ausdruck zusammengefügt werden.
- Jeder Operator hat eine definierte **Priorität** innerhalb der Klasse der Operatoren. Die Priorität legt fest, in welcher Reihenfolge ein zusammengesetzter Ausdruck berechnet wird.
- Jeder Operator hat eine **Bindungsrichtung**. Der Vorzeichenoperator - z.B. bindet den Ausdruck rechts von sich selbst, der Additionsoperator + bindet rechts und links.
- Für jeden Operator ist festgelegt, welchen Datentyp seine Operanden haben dürfen. So darf z.B. der Multiplikationsoperator * nicht auf Strings angewendet werden.

In C-Handbüchern finden sich auch die Bezeichnungen **Lwert** (lvalue) und **Rwert**, die die zulässigen Ausdrücke weiter klassifizieren. Lwerte sind Ausdrücke, die auf der linken Seite einer Zuweisung erscheinen können. Rwerte sind Ausdrücke, die nur auf der rechten Seite einer Zuweisung erscheinen können. Konstanten z.B. können keine Lwerte sein, da Konstanten nicht auf der linken Seite einer Zuweisung erscheinen können. Lwerte sind normalerweise Variablen (auch Arrayelemente und Strukturelemente mit Variablencharakter) oder Ausdrücke mit einem Sternoperator.

5.2 Die einzelnen Operatoren

5.2.1 Der Funktionsaufruf ()

Der Aufruf einer Prozedur wird durch runde Klammern hinter einem Prozedurnamen oder einem Prozedurpointer ausgedrückt. Zwischen der öffnenden und der schließenden Klammer dürfen aktuelle Prozedurargumente erscheinen. Die Aufrufklammern dürfen nicht mit den oben erwähnten Gruppierungsklammern verwechselt werden. In der Deklaration einer Prozedur werden ebenfalls runde Klammern verwendet. Details s. Kap. 10. Beispiele:

```
char *cfunct();    /* Dekl.: Funktion liefert (char *) */
int ifunct();      /* Dekl.: Funktion liefert int */
int (*pifunct)(); /* Dekl. eines Prozedurpointers */
```

```
cadr = cfunct();  /* Aufruf als Funktion */
cfunct();         /* Aufruf als Prozedur */
pifunct = ifunct; /* Zuweisung an Prozedurpointer */
(*pifunct)();     /* Aufruf ueber Prozedurpointer */
```

5.2.2 Auswahl eines Arrayelements [ ]

Ein Array ist eine Folge von C-Objekten des gleichen Typs, die über einen gemeinsamen Namen angesprochen werden können. Der Zugriff auf ein Einzelobjekt eines Arrays (ein Arrayelement) erfolgt durch Auswahl eines Elements über die Elementnummer. Die Elementnummer - der Arrayindex - wird in eckigen Klammern notiert. In der Deklaration eines Arrays dient die gleiche Notation zur Festlegung der Dimensionen eines Arrays. Hier folgt nur ein kurzes Beispiel. Größere Beispiele finden sich in Kap. 8.

```
/* Deklaration eines zweidimensionalen Arrays */
char carray[20][10],cv;
/* Zugriff auf ein Element des Arrays */
cv = carray[1][9];
```

5.2.3 Der Punktoperator: Auswahl eines Strukturelements

Eine Struktur ist eine Folge von Einzelvariablen mit unterschiedlichen Datentypen, die über den gemeinsamen Strukturnamen und den Elementnamen angesprochen werden können. Verbunde sind Sonderfälle von Strukturen. Details werden im Kapitel 9 erörtert. Hier nur ein kurzes Beispiel für Deklaration von Strukturen und Zugriff auf die Elemente. Bitte beachten Sie, daß auch der Pfeiloperator dem Zugriff auf Strukturelemente dient! Beide Operatoren können auch auf Verbunde angewandt werden.

```
/* Deklaration einer Struktur */
struct
{
  char name[20];
  long gehalt;
} pakte;
/* Zugriff auf die Elemente */
pakte.name[0] = 0;
pakte.gehalt  = 60000L;
```

5.2.4 Der Pfeiloperator: Auswahl eines Strukturelements

Wie der Punktoperator dient der Pfeiloperator zur Auswahl eines Strukturelements. Allerdings wird hier nicht ein Strukturnamen, sondern ein Pointer auf eine Struktur zum Einstieg in den richtigen Speicherbereich genutzt. Hier wieder nur ein kurzes Beispiel. Der gleiche Operator kann auch auf Verbunde angewandt werden. Details finden sich im Kap. 9.

```
/* Deklaration einer Struktur und eines Pointers */
struct
{
  char name[20];
  long gehalt;
} pakte,*ppakte;
/* Zugriff auf die Elemente */
ppakte = &pakte;
ppakte->name[0] = 0;
ppakte->gehalt  = 60000L;
```

5.2.5 Typumwandlung: Der Cast-Operator

Der Cast-Operator erzwingt die Herbeiführung eines gewünschten Typs. Er wird notiert als (*typ*) vor dem Objekt, das den gewünschten Typ erhalten soll. Typisches Beispiel: In C wie in den meisten anderen Programmiersprachen ist das Ergebnis einer Integerdivision eine *int*-Zahl. Ein Nachkommaanteil des Ergebnisses fällt also unter den Tisch. Um das zu verhindern, kann einer der beiden Operatoren (oder der Deutlichkeit halber beide) mit einem cast-Operator zwangskonvertiert werden:

```
printf("\n%f",(float)5 / 2);
printf("\n%f",5 / 2);  /* das ist falsch ! */
```

Das gleiche Ergebnis erhalten Sie, wenn Sie 5.0 / 2 schreiben. Bei Verwendung von *printf()* mit *%f* als Formatieranweisung ist die Übergabe von 5/2 als Argument falsch, weil *%f* besagt, daß eine *float*-Zahl übergeben wird, 5/2 aber in einem *int*-Ergebnis resultiert. Falsch wäre hier auch die Schreibweise (*float*)(5/2), weil dann nur das Ergebnis der Division konvertiert würde. Da die Integer-Division das Nachkommaergebnis ignoriert, nützt die Konvertierung des Ergebnisses nichts mehr!

Der Cast-Operator darf nicht dazu verwendet werden, die Größe eines Lwerts zu verändern. Das folgende Beispiel ist also nicht korrekt:

```
/* Beispiel ist nicht korrekt! */
short ivar;   /* soll 2 Bytes gross sein */
  (long)svar = 100000L;
```

5.2.6 Der sizeof-Operator

Der *sizeof*-Operator dient dazu, die Größe von Datenobjekten zu berechnen. Die errechnete Größe wird in *char*-Einheiten (also in Bytes) geliefert. *sizeof(int)* liefert den Wert 2, *sizeof(long)* den Wert 4, *sizeof(double)* den Wert 8 etc.

Der Ausdruck, dessen Größe errechnet werden soll, wird in Klammern hinter *sizeof* angegeben. Dabei sind zwei Varianten möglich: entweder wird nur ein Typ angegeben *(long, int, char*, über *typedef* definierte Typen ...), oder aber eine schon deklarierte Variable, ein Array, eine Struktur, ein Verbund etc.

sizeof ist keine Funktion! Der errechnete Wert wird vielmehr schon zur Compilezeit in den Code eingetragen. Wenn Sie also eine Pointervariable *cadr* definiert haben und *sizeof(cadr++)* schreiben, sollten Sie sich nicht wundern, daß *cadr* nicht erhöht wurde. Als wichtige Regel muß also festgehalten werden: *sizeof* nicht nebenbei zur Veränderung von Variablen verwenden!

Beispiele:

```
/* sizeof - Werte fuer Amiga */
char carray[10][10];
typedef
{
  int aaa;
  int bbb;
} MYSTRUCT;

sizeof(int) liefert Wert 2
sizeof(char) liefert Wert 1
sizeof(long) liefert Wert 4
sizeof(char *) liefert Wert 2   /* Aztec */
sizeof(char *) liefert Wert 4   /* Lattice */
sizeof(carray) liefert Wert 100
sizeof(carray[0]) liefert Wert 10
sizeof(MYSTRUCT) liefert Wert 4
```

5.2.7 Der Adreßoperator &

Der Adreßoperator & liefert die Adresse einer Variablen, eines Arrayelements oder einer Struktur. Arraynamen selbst sind schon Adressen, ebenso Funktionen. Auf Arraynamen und Funktionen sollte also der Adreßoperator nicht angewandt werden. Auf Registervariablen darf der Adreßoperator nicht angewandt werden, weil Maschinenregister nicht im normalen Speicher liegen und deshalb keine Adresse haben.

Als Regel gilt: Wenn der Adreßoperator & auf ein Objekt vom Typ *t* angewandt wird, liefert er eine Adresse vom Typ *t**.

```
/* Das folgende Programm enthaelt Fehler!!! */
int funktion(p1,p2)
int p1,p2;
{
register int hvar;
int *iadr;
char *cadr,carray[10];
  iadr = &hvar; /* verboten, weil hvar Registervariable  ist */
  iadr = &17;        /* falsch weil Konstante */
  cadr = &carray;    /* unnoetig - carray ist schon Adresse */
  cadr = &carray[0]; /* unnoetig kompliziert */
  cadr = carray;     /* == &carray[0]        */
  cadr = &carray[7]; /* Korrekt              */
  cadr = &funktion;  /* unnoetig, weil funktion Adresse ist */
  cadr = funktion;   /* korrekt              */
}   /* end funktion */
```

5.2.8 Der Sternoperator *

Der Sternoperator darf nicht mit der Multiplikation verwechselt werden, die ebenfalls mit * notiert wird. Während die Multiplikation ein zweistelliger Operator ist, ist der Sternoperator nur einstellig. Er ist das Gegenstück zum Adreßoperator &. Während der Adreßoperator nur auf Variablen (genauer Lwerte) angewandt werden kann, kann der Sternoperator auch auf Rwerte angewandt werden. Der Sternoperator macht aus einem Pointer wieder eine Variable. Beispiel:

```
int xyz;
int *axyz;

  xyz   = 2;      /* Normale Zuweisung */
  axyz  = &xyz;   /* Ermitteln der Adresse */
  *axyz = 3;      /* aequiv.: xyz = 3 */
  xyz   = *axyz;  /* aequiv.: xyz = xyz */
```

In dem Programmstück wird ein *int*-Variable *xyz* deklariert und ein Pointer auf Variablen vom Typ *int* namens *axyz*. Dann wird konventionell der Variablen *xyz* der Wert 2 zugewiesen. Danach weise ich der Pointervariablen *axyz* die Adresse von *xyz* zu (Adreßoperator &). Dann weise ich der Variablen *xyz* den Wert drei zu – diesmal unter Umweg über die Pointervariable *axyz*, deren Inhalt ja die Adresse von *xyz* ist! Die letzte Programmzeile ist eigentlich sinnlos: der Variablen *xyz* wird der Wert an der Adresse in *axyz* zugewiesen – also der Wert von *xyz* selbst. Wie Sie sehen, ist der Sternoperator auf der linken und auf der rechten Seite von Zuweisungen erlaubt.

5.2.9 Der Vorzeichenoperator -

Der Vorzeichenoperator dient dazu, negative Zahlen zu kennzeichnen. Sein Gebrauch ist aus der Schule bekannt. In C gibt es keinen einstelligen Plusoperator zur Kennzeichnung von positiven Zahlen!

```
Beispiele:
int aaa,bbb;
   aaa = -1;     /* Konstante */
   bbb = -aaa;   /* Variable  */
```

5.2.10 Die logische Negation !

Die logische Negation ist ein einstelliger Operator, der nur auf ganzzahlige Objekte angewandt werden darf. Sie liefert den Wert Eins, wenn der Operand Null ist und den Wert Null, wenn der Operand ungleich Null ist. Beispiele:

```
a = 17;
b = ! a;    /* b wird 0 */
a = 0;
b = ! a;    /* b wird 1 */
```

5.2.11 Der Komplementoperator ~

Der einstellige Komplementoperator darf nur auf ganzzahlige Objekte angewandt werden. Er invertiert das Bitmuster des rechts von ihm stehenden Operanden. Dieser Operator ist das bitweise Gegenstück zum logischen Nicht-Operator. Beispiel:

```
a = 0xffff;    /* 1111 1111 1111 1111 */
b = ~a;        /* 0000 0000 0000 0000 */
a = 0x5555;    /* 0101 0101 0101 0101 */
b = ~a;        /* 1010 1010 1010 1010 */
```

5.2.12 Der Additionsoperator +

Der Additionsoperator + ist zweistellig. Er dient dazu, Zahlen vom Typ *char, int, long, float* und *double* zu addieren. Er ist auch auf die *unsigned*-Varianten dieser Typen anwendbar. Bitte beachten Sie, daß Pointer normalerweise nicht addiert werden dürfen. Die von mir getesteten Compiler reagierten unterschiedlich streng auf entsprechende Versuche. Gemischte Ausdrücke aus Pointern und ganzzahligen Objekten können mit dem Additionsoperator gebildet werden. Auch Strings können nicht über den Additionsoperator verknüpft werden, wie Sie es vielleicht aus BASIC kennen.

Beispiel Addition gemischte Objekte:

```
int *iadr,ivar;
    iadr = &ivar;
    iadr = iadr + 4;    /* gemischter Ausdruck */
    printf("\n&ivar= %lx iadr= %lx",&ivar,iadr);

Ergebnis(auf VAX, sizeof(int) == 4):
    &ivar = 7ff2f17c iadr = 7ff2f18c
```

5.2.13 Der Subtraktionsoperator -

Der Subtraktionsoperator – ist zweistellig. Er dient dazu, Zahlen der Typen *char, int, float* und *double* sowie der entsprechenden *unsigned* Varianten zu subtrahieren. Im Gegensatz zur Addition darf die Subtraktion auf Pointer angewandt werden, sofern dies sinnvoll ist.

Bei der Subtraktion gemischter Objekte gelten die gleichen Regeln wie bei der Addition gemischter Objekte.

5.2.14 Der Multiplikationsoperator *

Die Multiplikation ist – im Gegensatz zum Sternoperator – zweistellig. Die Multiplikation ist nicht auf Pointer anwendbar. Da der Multiplikationsoperator vom Aussehen her mit dem Sternoperator identisch ist, sollte er links und rechts durch Blanks begrenzt werden. Für besonders schnelle Berechnungen kann die Multiplikation durch den Links-Shiftoperator (s. 5.2.18) ersetzt werden.

5.2.15 Der Divisionsoperator /

Die Division ist eine zweistellige Operation. Wie in den meisten höheren Programmiersprachen liefert eine Division zweier ganzzahliger Größen einen ganzzahligen Wert. 5/2 ergibt also den Wert 2 und nicht 2.5! Division durch Null ist wie üblich verboten und führt normalerweise zum Programmabsturz. Die Division ist nicht auf Pointer anwendbar.

5.2.16 Der Restoperator %

Der Restoperator (auch Modulo-Operator genannt) ist ein zweistelliger Operator, der nur auf ganze Zahlen angewandt werden kann. Er liefert den ganzzahligen Restwert, der bei der Division des ersten Operanden durch den zweiten Operanden entsteht. Der Ausdruck 5/2 liefert der Wert 2, der Ausdruck 5%2 den Wert 1. Es gilt also die Beziehung:

```
((op1 / op2) * op2) + (op1 % op2) == op1
```

Neben der eigentlichen Funktion – der Errechnung des Restwerts – kann der Operator % auch für sonstige Aufgaben verwendet werden, z.B. zur Prüfung, ob ein Ausdruck ungeradzahlig ist: *if (var1 % 2)* .....

5.2.17 Der Shiftoperator >>

Der Shiftoperator >> shiftet (schiebt) das im ersten Operanden enthaltene Bitmuster um die Anzahl Stellen nach rechts, die im zweiten Operator angegeben ist. Er darf normalerweise nur auf ganzzahlige Typen angewandt werden. Der zweite Operand sollte einen *unsigned*-Datentyp haben. Das Vorzeichenbit eines vorzeichenbehafteten Objekts wird mitverschoben. Links frei werdende Stellen werden in vorzeichenbehafteten Objekten mit dem Vorzeichenbit aufgefüllt. Sehr oft wird der Operator >> zur schnellen Division verwendet. So ergibt z.B. 256>>1 den Wert 128. Diese Art der Division funktioniert nur als Ersatz für Divisorwerte, die Zweierpotenzen darstellen.

Beispiele:

```
(-1) >> 1                    == 0xffffffff
0x80000000 >> 1              == 0xc0000000
(unsigned)0x80000000 >> 1    == 0x40000000
1    >> 1                    == 0x00000000
0x100 >> 2                   == 0x00000040
```

5.2.18 Der Shiftoperator <<

Der Shiftoperator << shiftet (schiebt) das im ersten Operanden enthaltene Bitmuster um die Anzahl Stellen nach links, die im zweiten Operator angegeben ist. Er darf normalerweise nur auf ganzzahlige Typen angewandt werden. Durch Shiften nach links kann aus einem positiven Wert ein negativer Wert werden, wenn in das Vorzeichenbit eine Eins geschoben wird. Der zweite Operator sollte einen *unsigned*-Datentyp haben. Sehr oft wird der Operator << zur schnellen Multiplikation verwendet. So ergibt z.B. 256<<2 den Wert 1024. Diese Art der Multiplikation funktioniert nur als Ersatz für Multiplikatorwerte, die Zweierpotenzen darstellen. Beispiele:

```
(-1) << 1          == 0xfffffffe
0x100  << 2        == 0x00000400
0x40000000 << 1 == 0x80000000 == -2147483648
```

5.2.19 Der logische Oder-Operator ||

Der logische Oder-Operator ist zweistellig und wird auf ganzzahlige Objekte angewandt. Der Ausdruck *(op1 || op2)* liefert den Wert 1 (ganzzahlig), wenn mindestens einer der beiden Operanden ungleich Null ist – sonst wird Null als Wert geliefert. Dieses Oder wird auch 'inklusives Oder' genannt im Unterschied zum exklusiven Oder, dem EXOR. Im Gegensatz zum EXOR liefert ODER auch dann einen Wert ungleich Null (also TRUE), wenn beide Operanden wahr sind. EXOR liefert dann den Wert 0. Bitte beachten Sie, daß es in C keinen Datentyp wie *Boolean* in Pascal gibt. Jeder ganzzahlige Wert ungleich Null gilt als TRUE (wahr); der Wahrheitswert FALSE ist gleich der binären Null.

5.2.20 Der logische Und-Operator &&

Der logische Und-Operator ist zweistellig und wird auf ganzzahlige Objekte angewandt. Der Ausdruck *op1 && op2* liefert den Wert 1, wenn beide Operanden ungleich Null sind. In allen anderen Fällen wird Null geliefert.

5.2.21 Der bitweise Oder-Operator |

Der bitweise Oder-Operator verknüpft jedes Bitpaar der ganzzahligen beiden Operanden über ein inklusives Oder. Während der logische Oder-Operator nur 0 oder 1 lieferte, kann der bitweise Oder-Operator deshalb alle Bitkombinationen liefern, die in ganzzahligen Datenobjekten denkbar sind. Die Oder-Operation ergibt den Wert Wahr (hier: gesetztes Bit), wenn mindestens einer der Operanden Wahr ist. Beispiel:

```
a = 0x55;  /* 0000 0000 0101 0101 */
b = 0xaa;  /* 0000 0000 1010 1010 */
c = a | b; /* 0000 0000 1111 1111 = 0xff */
```

5.2.22 Der bitweise Und-Operator &

Der bitweise Und-Operator verknüpft jedes Bitpaar der beiden ganzzahligen Operanden durch ein Und. Der Und-Operator liefert Wahr (setzt ein Bit), wenn beide Operanden wahr sind (beide Operanden an der jeweiligen Position ein gesetztes Bit aufweisen). Dieser Operator wird oft zum Ausmaskieren von Bits verwendet. Ein bitweises Und mit *0x7f* erzeugt z.B. ASCII-konforme Zeichen. Beispiel:

```
a = 0x55;   /* 0000 0000 0101 0101 */
b = 0x4f;   /* 0000 0000 0100 1111 */
c = a & b;  /* 0000 0000 0100 0101 = 0x45 */
```

5.2.23 Der bitweise Exor-Operator ^

Es gibt nur den bitweisen Exor-Operator – eine logische Variante des EXOR existiert nicht. Der zweistellige bitweise EXOR Operator kann nur auf ganzzahlige Objekte angewandt werden. EXOR ergibt den Wert Wahr, wenn genau einer der beiden Operanden Wahr ist. Beispiel:

```
a = 0x55;   /* 0000 0000 0101 0101 */
b = 0x4f;   /* 0000 0000 0100 1111 */
c = a ^ b;  /* 0000 0000 0001 1010 = 0x1a */
```

5.2.24 Der Gleichheitsoperator ==

Der Gleichheitsoperator darf nicht mit dem Zuweisungsoperator = verwechselt werden. Der Gleichheitsoperator kann nur auf einfache oder abgeleitete Datentypen angewandt werden. Er liefert den Wert Eins im Fall der Gleichheit und sonst den Wert Null. Beispiele:

```
if (a == c) d = e;
if (fvar == 2.5) fvar = 10.3e4;
if (pchar == NULL) pchar = &carray[17];
iarray[a == c] = x;
```

5.2.25 Der Ungleichheitsoperator !=

Der Ungleichheitsoperator darf nur auf einfache oder abgeleitete Datentypen angewandt werden. Er liefert den Wert Eins im Fall der Ungleichheit und sonst den Wert Null.

```
if (a != c) d = e;
if (fvar != 2.5) fvar = 10.3e4;
if (pchar != NULL) pchar = &carray[17];
iarray[a != c] = x;
```

Den Ungleichheitsoperator können Sie durch die Kombination von logischer Negation und Gleichheitsoperator ersetzen wie im folgenden Beispiel:

```
/* mit Ungleichheitsoperator */
if (a != b) c = d;
/* umformuliert */
if (! (a == b)) c = d;
```

5.2.26 Die Zuweisungsoperatoren

Es gibt in C mehrere Zuweisungsoperatoren. Der wichtigste Zuweisungsoperator ist =; alle anderen Zuweisungsoperatoren können als Abkürzungen für Kombinationen aus = mit anderen arithmetischen Operatoren verstanden werden.

Der Zuweisungsoperator darf in K&R nur auf einfache und abgeleitete Datentypen angewandt werden. Er liefert als Wert das zuletzt zugewiesene Objekt. Er bindet nach rechts. Damit läßt sich die Zulässigkeit von Mehrfachzuweisungen wie im folgenden Beispiel erklären:

```
a = b = c = d = 5;
/* wird so abgearbeitet */
a = (b = (c = (d = 5)));
```

Liste der kombinierten Zuweisungsoperatoren:

Operator	Beispiel	Bedeutung
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
>>=	a >>= b	a = a >> b
<<=	a <<= b	a = a << b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

Der Zuweisungsoperator darf nicht auf Arrays angewandt werden. In der C-Bibliothek gibt es die Funktionen *strcpy()* und *strncpy(),* die für die Zuweisung von *char*-Arrays verwendet werden können. Andere Arrays können leicht über eigene Prozeduren oder in Schleifen elementweise zugewiesen werden. In der K&R-Version darf der Zuweisungsoperator auch nicht auf Strukturen und Verbunde angewandt werden. Eine Zuweisung der Form

```
pchar = "ABC";
```

ist nicht etwa die Zuweisung eines Arrays, sondern die Zuweisung der Adresse des Strings "ABC" an die Pointervariable *pchar.* Die Adresse des Strings "ABC" ist eine Pointerkonstante.

5.2.27 Der Fragezeichenoperator ?

Der Fragezeichenoperator ist der einzige dreistellige Operator in C. Er enthält eigentlich zwei Teiloperatoren, die durch ? und : notiert werden. Allgemeine Form:

```
ausdruck1 ? ausdruck2 : ausdruck3
```

Der Operator liefert als Wert *ausdruck2* wenn *ausdruck1* ungleich Null ist und sonst *ausdruck3.* Die Übersetzung in einen *if-else* Befehl sieht am Beispiel so aus:

```
/* mit Fragezeichenoperator */
return(((a + b) > c) ? d : e);
/*  uebersetzt */
if ((a + b) > c) return(d);
else             return(e);
```

5.2.28 Der Kommaoperator ,

Der Kommaoperator dient dazu, die Evaluierungsreihenfolge von Ausdrücken festzulegen. Es gibt in C keine Vorschrift, in welcher Reihenfolge gleichberechtigte Ausdrücke berechnet werden sollen. Der Kommaoperator legt eine Links-Rechts-Abarbeitungsreihenfolge fest. Das ist wichtig für Fälle, in denen der Wert eines Ausdrucks von der Berechnung eines anderen Ausdrucks abhängt wie im folgenden Beispiel. Der Wert eines mit dem Kommaoperator gebildeten Ausdruck ist der Wert des rechtesten Teilausdrucks:

```
    var1 = (var2 = 1, var2 += 2);
Ergebnis: var1 ist 3
          var2 ist 3
```

Der Kommaoperator wird besonders häufig in der Schleifenliste der *for*-Schleife verwendet.

5.3 Vorrangregeln für Operatoren

C-Ausdrücke können sich aus mehreren Teilausdrücken zusammensetzen, die durch Operatoren verknüpft werden. Wenn die zu einem Ausdruck zusammengeschlossenen Einzelausdrücke durch Gruppierungsklammern markiert sind, respektiert der Compiler diese Markierung bei der Codegenerierung; die Einzelausdrücke werden dann zur Laufzeit in der markierten Hierarchie berechnet (evaluiert). Die Berechnung beginnt dann bei den am tiefsten eingebetteten Ausdrücken. Ein Beispiel für einen Ausdruck mit Gruppierungsklammern:

```
ivar = iarray[((a + b) / (c - d))];
```

Die Ausdrücke *(a + b)* und *(c - d)* werden hier zuerst berechnet. Erst dann wird dividiert. Diese Gruppierungsklammern aber müssen nicht sein! Schon in der Schule haben Sie Regeln gelernt wie 'Punktrechnung geht vor Strichrechnung', also Vorrangregeln (Prioritätsregeln). Da C sehr viele Operatoren hat, gibt es auch eine Menge von Vorrangregeln. Ich führe die Hierarchie der Operatoren nachstehend auf. Vorab aber eine Warnung: Sie werden sich diese Regeln kaum merken können. Deshalb der Rat, in Zweifelsfällen lieber Klammern benutzen als auf Vorrangregeln zu vertrauen! Die nachstehend zuerst genannten Operatoren haben die höchste Priorität, d.h. die Teilausdrücke in denen sie stehen, werden zuerst berechnet.

Name Operator	Symbol
Funktionsaufruf	( )
Arrayelement	[]
Pfeiloperator	->
Punktoperator	.
log. Negation	!
bitweise Negation	~
Inkrementoperator	++
Dekrementoperator	--
Negativvorzeichen	-
cast - Operator	(typ)
Sternoperator	*
Adressoperator	&
sizeof - Operator	sizeof
Multiplikation	*
Division	/
Restoperator	%
Addition	+
Subtraktion	-
Shift links	<<
Shift rechts	>>
Kleiner als	<
Kleiner gleich	<=
Groesser als	>
Groesser gleich	>=
Gleichheit	==
Ungleichheit	!=
Bitweises Und	&
Bitweises Exklusives Oder	^
Bitweises Oder	|
Logisches Und	&&
Logisches Oder	||
Fragezeichenoperator	&
Zuweisung(mit Varianten)	=
Kommaoperator	,

Das folgende Beispiel gibt die Berechnungsreihenfolge für einen Ausdruck an, der keine Gruppierungsklammern enthält. Versuchen Sie bitte, anhand der vorstehenden Tabelle die Berechnungsreihenfolge nachzuvollziehen:

```
/* iarray[1] hat den Wert 3 */
~ 4 + 8 * iarray[1]

1.  iarray[1] : 3
2.  ~ 4       : -5
3.  8 * 3     : 24
4.  -5 + 24   : 19

/* aequivalenter Ausdruck */
(~ 4) + (8 * (iarray[1]))
```

6 Befehle

6.1 Befehlstypen

Programme bestehen aus Befehlen, die üblicherweise in der niedergeschriebenen Reihenfolge ausgeführt werden. In C-Programmen sollte man zwei Typen von Befehlen unterscheiden:

- Befehle für den Präprozessor
- C-Befehle

Der Präprozessor ist bekanntlich nicht Bestandteil der Sprache C, findet sich aber praktisch in allen Implementierungen. Befehle für den Präprozessor beginnen immer mit dem Nummernzeichen #. Der eigentliche Compiler sieht diese Befehle nicht mehr – sie sind vom Präprozessor aufgelöst worden. Ich behandle die Präprozessorbefehle im Kapitel 11. Hier geht es also nur um die eigentlichen C-Befehle, die der Compiler sieht. Hier sind drei Typen zu unterscheiden:

- Befehle ohne Schlüsselwörter, z.B. Zuweisungen, Prozeduraufrufe. Solche Befehle bestehen aus beliebigen C-Ausdrücken.
- Befehle, die mit einem Befehlsschlüsselwort beginnen.

Es gibt noch andere Möglichkeiten der Klassifikation von C-Befehlen: Befehle sind einfach oder zusammengesetzt, ausführbar oder nicht ausführbar. Die Unterscheidung ausführbar – nicht ausführbar geht davon aus, daß den ausführbaren Befehlen im übersetzten Programm eine bestimmte Codestrecke entspricht, die zur Laufzeit ausgeführt wird. Die nicht ausführbaren Befehle (Deklarationen) werden bisweilen nicht als 'Befehle' bezeichnet. Aus der Sicht des Programmierers jedoch und auch syntaktisch stellen sie normale Befehle dar; deshalb werden sie in diesem Kapitel behandelt.

In C können sich Befehle (anders als in BASIC) über mehrere Zeilen erstrecken. Sie dürfen an beliebiger Stelle in der Zeile beginnen und aufhören. Sie benötigen (anders als in Fortran) kein besonderes Kennzeichen für die Befehlsfortsetzung. Einfache Befehle werden mit einem Semikolon (";") abgeschlossen. Zusammengesetzte Befehle beginnen mit einer öffnenden und enden mit einer schließenden geschweiften Klammer. Zwischen den geschweiften Klammern (entsprechend dem *begin* und *end* in Pascal) stehen dabei weitere einfache oder zusammengesetzte C-Befehle. Zusammengesetzte Befehle (compound statements) bilden **Blöcke**. In solchen Blökken können lokale Variablen deklariert werden, die nur innerhalb dieses Blocks gültig sind. Hier einige Beispiele für Befehle in C:

```
/* Einige einfache Befehle */
int ivar1,ivar2;          /* Deklaration */
a = b + c;                /* Zuweisung */
proc1(b,c);               /* Prozeduraufruf */
a++;                      /* auch eine Zuweisung */
if (a == b) c = a * b;    /* Schluesselwort if */
if (a != b)               /* ueber mehrere Zeilen */
            c = a / b;    /* verteilter Befehl    */

/* Ein zusammengesetzter Befehl */
if (a == b)
{
int c;                    /* lokale Blockvariable */
  c = a * b;
  a = b / c;
}                         /* Kein Semikolon */
```

Übrigens würde ein Semikolon hinter dem zusammengesetzten Befehl nicht stören. Er würde als der leere Befehl interpretiert, der in C tatsächlich zulässig ist und auch vorkommt, z.B. in *for*-Schleifen.

6.2 Deklarationen

Die Deklaration kann als nicht ausführbarer Befehl betrachtet werden. Sie legt primär Datentyp, Speicherklasse und Gültigkeitsbereich von Variablen, Strukturen oder Funktionen fest. Daneben erlaubt die Deklaration in C auch das Initialisieren (Vorbesetzen) der deklarierten Objekte. Die Deklaration hat in C die folgende allgemeine Form:

```
[bereich] [klasse] [typ] name [= vorbesetzung]
                          [, name [= vorbesetzung]] ...;
```

Die eckigen Klammern sind hier Optionalklammern, d.h. sie umschließen Elemente, die erscheinen können aber nicht erscheinen müssen. Eine Deklaration muß also

mindestens aus einem Namen bestehen. Sie kann eine Vorbesetzung aufweisen; sie kann außerdem Attribute wie Gültigkeitsbereich, Speicherklasse und Datentyp (Standardannahme ist *int*) aufweisen. In einem Deklarationsbefehl können mehrere Objekte deklariert werden, die dann durch Kommata getrennt werden. Die Konzepte Gültigkeitsbereich, Speicherklasse und Datentyp sind bereits an anderer Stelle besprochen worden. Auf die Deklaration und Initialisierung von Arrays, Strukturen, Verbunden und Funktionen wird später eingegangen. Hier einige Beispiele für Deklarationen:

```
/* Deklarationen von Objekten mit einfachen Typen */
int var1,var2,var3;
char cvar1,cvar2;

/* Deklarationen abgeleiteter Typen */
int *pvar1,*pvar2;
char *pcvar1,*pcvar2;

/* Deklarationen zusammengesetzter Datentypen */
int iarray[10];
char carray[20];
char *carray[12];

/* Weglassen der Dimension - z.B. bei einem */
/* Prozedurargument                         */
char carray[];

/* Angabe des Gueltigkeitsbereichs */
extern int var1,var2,var3;

/* Angabe von Speicherklassen */
static char cvar1,cvar2;
register char *pcvar1,pcvar2;

/* Deklaration einer Funktion */
extern float wurzel();
/* Pointer auf eine Funktion */
float (*pfunct)();

/* einfache Initialisierungen */
int var1 = 10, var2 = 20;
char cvar1 = 'a', cvar2 = 'b';

/* Initialisierungen von Arrays */
int iarray[3] = { 1,2,3 } ;
```

```
/* Stringarray: Platz fuer die binaere Null lassen! */
char carray[4] = "abc";

/* Initialisierung Zweidimensionaler Array */
int iarray[3][2] = { {1,2,3} , {4,5,6} };
int iarray[3][2] = { 1,2,3 , 4,5,6};

/* Initialisierung einer Struktur */
struct
{
  int a,b,c;
  char carray[4];
  char *next;
} mystruct =
  {
    { 1,2,3};
    { "abc" };
    { NULL }
  };
```

6.3 Zuweisungen

Die Zuweisung ist ein C-Ausdruck, der auch als Befehl verwendet werden kann. Im Kapitel 4 wurde bereits erwähnt, daß Zuweisungen in C wie Operatoren wirken, d.h. sie produzieren einen Wert. Wie in jeder anderen höheren Programmiersprache bildet der Zuweisungsoperator mit seinen Operanden einen Befehl, z.B.:

```
a = b + c;
```

Sie dürfen keinesfalls den Gleichheitsoperator == mit dem Zuweisungsoperator = verwechseln.

Die linke Seite einer Zuweisung muß aus einem Lwert (lvalue) bestehen, also aus einem Ausdruck, dem ein Wert zugewiesen werden kann. Ein Lwert ist z.B. eine Variable, ein Arrayelement, ein Strukturelement, ein Ausdruck mit einem Sternoperator. Beachten Sie besonders, daß es in C in der K&R Version keine Zuweisungen an Arrays, Strukturen und Verbunde gibt, außer in Initialisierungen. Dafür müssen Funktionen verwendet werden, die zum Teil in der C-Bibliothek vordefiniert sind, z.B. *strcpy()*. Es können natürlich auch eigene Funktionen geschrieben werden oder aber Schleifen eingesetzt werden, in denen die Zuweisung elementweise geschieht.

Zuweisungsbefehle in C haben mehr Varianten als in anderen Programmiersprachen, weil es mehr Zuweisungsoperatoren gibt. Die vollständige Liste der Zuweisungsoperatoren ist in Kap. 4 wiedergegeben. Deshalb sollen nur zwei weitere Zuweisungsbefehle stellvertretend für alle anderen angegeben werden:

```
a += b;   /* a = a + b */
a--;      /* a = a - 1 */
```

Der Befehl *a += b* ist die abkürzende Schreibweise für *a = a + b*. Der zweite Befehl *a--* ist ebenfalls eine abkürzende Schreibweise, und zwar für *a = a - 1*. Im Befehl wird hier nicht der Zuweisungsoperator, sondern der Dekrementoperator verwendet. Wie Sie sehen, läßt sich die Vielfalt der Zuweisungsbefehle in C problemlos auf einen einzigen Typ reduzieren.

6.4 Prozeduraufrufe

Wie die Zuweisung ist auch der Prozeduraufruf ein Ausdruck, der als Befehl verwendet werden kann. Die Prozeduraufrufe in C werden durch ein Paar runder Klammern (dazwischen eventuell Argumente, die durch Kommata getrennt werden) syntaktisch ausgedrückt. Davor muß ein Prozedurname oder ein Pointer auf eine Prozedur erscheinen. Wenn eine Prozedur als Funktion verwendet wird, so erscheint der Aufruf auf der rechten Seite einer Zuweisung, als Operand eines Operators, oder im Argument einer anderen Prozedur. Damit ist nur die Verwendung als Unterprogramm (nicht als Funktion) ein selbständiger Befehl. Funktionswerte werden über den *return*-Befehl übergeben. Prozeduren werden im Detail im Kapitel 9 behandelt. Deshalb hier nur eine Übersicht über ihre syntaktische Verwendung:

```
/* Unterprogramm: Aufruf ohne Argumente */
proc1();
/* Unterprogramm: Aufruf mit Argumenten */
proc2(a,b,c);
/* Aufrufe als Funktion */
a = proc3(b,c);
printf("\nWert proc3= %d",proc3(b,c));
if (proc3(b,c) == proc3(a,b)) ....
```

6.5 Bedingungen

In C gibt es drei Konstruktionen, die Bedingungen formulieren: den *if*-Befehl, den *switch*-Befehl und den ?-Operator (s. Kap. 4). Ein mit dem ?-Operator (s. Kap. 4) gebildeter Ausdruck ist als Befehl allein lebensfähig wie in dem folgenden Beispiel:

```
(a > b) ? (c = 1) : (c = 2);
/* Bedeutung in if - Befehl uebersetzt: */
/* if (a > b) c = 1;                    */
/* else       c = 2;                    */
```

Dennoch finden sich Befehle mit dem ?-Operator nur selten in C-Programmen, wohl weil sie nicht sehr eingängig sind. Wie im Beispiel angegeben läßt sich ein mit dem

?-Operator gebildeter Befehl in eine besser lesbare *if-else*-Konstruktion übersetzen. Häufiger findet man Ausdrücke mit dem ?-Operator in Makros und als Bestandteil anderer Befehle.

Der *if*-Befehl besteht aus zwei Teilen: dem *Wenn*-Teil (Bedingung) und dem *Dann*-Teil. Der *Dann*-Teil besteht aus einem vollgültigen C-Befehl. Der *if*-Befehl hat die allgemeine Form:

```
if ( Bedingung ) Befehl;
```

Diese Form entspricht nahezu komplett der Syntax anderer Programmiersprachen (z.B. Pascal). Auf zwei kleine Unterschiede muß hingewiesen werden: ein Kennwort wie *THEN* gibt es in C nicht und das abschließende Semikolon darf bei einem einfachen Befehl im *Dann*-Teil nicht fehlen – selbst wenn ein *else*-Befehl folgt.

Die Bedingung ist ein C-Ausdruck, der zu einem ganzzahligen Wert berechnet wird. Wenn der berechnete Wert ungleich Null ist, wird der folgende Befehl (kann auch ein zusammengesetzter Befehl zwischen geschweiften Klammern sein) ausgeführt. Ein leerer Befehl (nur Semikolon) im Dann-Teil des *if*-Befehls ist zulässig.

Auf den *Dann*-Teil eines *if*-Befehls kann ein *else*-Befehl folgen, der aus dem Kennwort *else* und einem (eventuell zusammengesetzten) C-Befehl besteht. Ein *else*-Befehl wird nur dann ausgeführt, wenn der *Wenn*-Teil des vorhergehenden *if*-Befehls den Wert Null ergab, wenn also der *Dann*-Teil des *if*-Befehls nicht ausgeführt werden konnte. Im *else*-Befehl kann selbst wieder ein *if*-Befehl erscheinen. Beispiele für *if*-Befehle:

```
/* Bedingung ist Variablenwert     */
/* Einfacher Befehl im Dann - Teil */
if (test) printf("\nStelle 17 erreicht");

/* Bedingung ist ein Vergleichsausdruck    */
/* Zusammengesetzter Befehl im Dann - Teil */
if (a > b)
{
  c = a - b;
  printf("\nDifferenz a-b = %d",c);
}

/* if - Befehl mit else - Befehl */
if ((ifil = fopen("test.c","r")) != NULL)
{
  printf("\n--- test.c geoeffnet ---");
  verarbeitung(ifil);
}
```

```
else
{
  printf("\n--- Dateifehler ---");
  exit(1);
}
```

Der *switch*-Befehl ("Schalter-Befehl") ist dem *if*-Befehl sehr verwandt. Während allerdings der *if*-Befehl nur zwei Teile hat, besteht der *switch*-Befehl meist aus mehreren Teilen. Er ist praktisch immer ein zusammengesetzter Befehl und hat den allgemeinen Aufbau:

```
switch (Auswahlkriterium) [Fall1 Fall2 ... Falln Sonstfall]
```

Die eckigen Klammern sind hier wieder Optionalklammern. Als Auswahlkriterium wird nach dem Schlüsselwort switch ein ganzzahliger Ausdruck angegeben, der zur Fallunterscheidung herangezogen wird. Jeder Fall besteht aus einer **Fallmarke** (label) und einem (eventuell leeren) Befehl, der zusammengesetzt sein kann. Die Fallmarken werden durch das Schlüsselwort *case* oder *default* (im Sonstfall) eingeleitet. Hinter *case* erscheint ein ganzzahliger Konstantenausdruck. Jede Fallmarke wird durch einen Doppelpunkt beendet. In allen Fällen darf das Schlüsselwort *break* verwendet werden, um den aktuellen Fall und den *switch*-Befehl zu verlassen. In speziellen Anwendungen kann ein Fall auch durch *return*, *goto* oder *exit()* verlassen werden.

Sie sehen schon, daß der *switch*-Befehl der komplexeste C-Befehl ist. Er ist im allgemeinen ein zusammengesetzter Befehl und seine Fälle bestehen meist auch aus zusammengesetzten Befehlen. Vier Schlüsselwörter (*switch, case, default, break*) gehören zu seinem syntaktischen Umfeld.

Besser als viele Erklärungen sind wieder Beispiele. Das folgende Beispiel zeigt eine einfache Fallunterscheidung mit mehreren Fällen und einem Sonstfall. Das Programm liest ein einbuchstabiges Kommando ein und ruft die dazugehörenden Kommandoprozeduren auf. Die Kommandos sollen klein oder groß eingegeben werden:

```
/****************************************

                Switch-Test
            last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
        (C) 1987  by Markt & Technik

****************************************

Testprogramm fuer den Switch-Befehl

****************************************/

#include <stdio.h>

lesen()
{
  printf("\nIn Lesen()\n");
}   /* end lesen */

schreiben()
{
  printf("\nIn Schreiben()\n");
}   /* end schreiben */

ausfuehren()
{
  printf("\nIn Ausfuehren()\n");
}   /* end ausfuehren */

main()
{
  char hfe[80];

  while (1)
    {
     printf("\nL>esen</S>chreiben</A>usfuehren</E>nde< :");
     gets(hfe);

     switch (hfe[0])
       {
        case 'l':
        case 'L : lesen();
                  break;
        case 's':
        case 'S': schreiben();
                  break;
        case 'a':
        case 'A': ausfuehren();
                  break;
        case 'e':
        case 'E': exit(0);
        /* Ansonsten */
        default: printf("--- Falsches Kommando ---");
       }  /* end switch  */
    }  /* while 1 */
}  /* end main */
```

Das Programm hätte mit einem einfachen Aufruf von *toupper()* auch einfacher (mit weniger Marken) formuliert werden können. Diese Version jedoch erlaubt mir den Hinweis, daß vor einem Fall mehrere Marken erscheinen können. Die *break*-Befehle sind nötig, damit die Programmausführung nicht in den nächsten Fall 'weiterrutscht". Im Fall für das Ende-Kommando ist *break* nicht nötig, weil *exit()* sowieso einen Abbruch des Falles und des kompletten Programms bewirkt. Der Sonstfall (er hat die Marke *default*:) wird dann ausgeführt, wenn keiner der vorher genannten Fälle zutrifft. Auch er wird hier nicht mit *break* beendet, weil die Programmausführung sowieso auf das Ende des *switch*-Befehls läuft.

switch-Befehle sind nur dann wirklich sinnvoll, wenn mehr als drei Fälle unterschieden werden. Andernfalls ergeben *if*-Befehle kürzeren und schnelleren Code. Aber das sollten Sie im Bedarfsfall an Ihrer Maschine testen.

Ich sagte schon, daß der *switch*-Befehl ein komplexer Befehl ist. Sie sollten deshalb darauf gefaßt sein, beim Debuggen (z.B. wenn Sie einen Breakpoint setzen wollen) zuerst einmal ratlos im Binärprogramm zu suchen. Die Compiler lösen den *switch*-Befehl nicht einfach in *if*-Befehle auf; vielmehr wird eine Sprungleiste aufgebaut, die anhand des Auswahlkriteriums zur Errechnung des Sprungbefehls verwendet wird. Wenn Sie einen Source-Level Debugger haben, brauchen Sie sich darum keine großen Gedanken zu machen, da dann Breakpoints auf der Ebene von C-Befehlen gesetzt werden können.

Umsteiger aus anderen Programmiersprachen sollten daran denken, daß der *switch*-Befehl in anderen Sprachen mit *case* oder *select* eingeleitet wird und meist eine etwas andere Syntax hat: der *break*-Befehl wird entweder nicht erlaubt oder ist implizit (ohne Nennung) vorhanden. Denken Sie auch daran, daß *switch* und nicht *case* den Befehl einleitet: *case* hat hier nur die Funktion, eine Fallmarke zu kennzeichnen.

6.6 Schleifen

In C gibt es drei Schleifenkonstrukte:

- die *for*-Schleife
- die *while*-Schleife
- die *do while*-Schleife

Auch mit *goto* und Sprungmarken können Schleifen formuliert werden, die jedoch dem üblichen C-Stil widersprechen. Die einfachste Schleife ist die *while*-Schleife. Sie hat die allgemeine Form:

- while (Schleifenbedingung) [Schleifenkoerper];

Der Schleifenkörper kann leer sein (nur Semikolon); im Normalfall besteht er aus einem einfachen oder einem zusammengesetzten C-Befehl (einem Block). Die in run-

den Klammern stehende Schleifenbedingung muß aus einem Ausdruck bestehen, der zu Null (= Falsch) oder ungleich Null (= Wahr) evaluiert werden kann. Die Schleifenbedingung wird überprüft, **bevor** der Schleifenkörper ausgeführt wird. Bitte beachten Sie, daß im Gegensatz zur *for*-Schleife eventuell vorhandene Schleifenzähler zum Schleifenkörper gehören. In keinem der C-Schleifenkonstrukte gibt es einen Mechanismus zum automatischen Hochzählen von Schleifenvariablen; dafür muß der Programmierer selbst sorgen. Beispiele für *while*-Schleifen:

```
/* leerer Schleifenkoerper */
a = 0;
while (a++ < 10) ;

/* einfacher Befehl als Schleifenkoerper */
a = 0;
b = 0;
while (a++ < 10)
  b = b + a;

/* Zusammengesetzter Befehl als Schleifenkoerper */
a = b = 0;
while (a < 10)
{
  b = b + a;
  a = a + 2;
}
/* Eine Endlosschleife: Die Schleifenbedingung ist */
/* eine Konstante                                  */
while (1)
  gets(puffer);

/* Der Schleifenkoerper wird u.U. nicht durchlaufen */
/* (Falls das gelesene a den Wert 0 hat! )          */
scanf("%d",&a);
while (a)
  b = b + a--;
```

Um dem Programmierer Sprünge zu ersparen, wurden in C die Befehle *break* und *continue* aufgenommen. Mit *break* kann die aktuelle Schleife verlassen werden. Mit *continue* können Sie ans Ende der aktuellen Schleife springen. Ein Beispiel für *break* und *continue*:

```
/* Filekopie mit Ignorieren von Leerzeilen */
while (1)
{
  /* Aussteigen aus der Schleife bei EOF */
```

```
  if (fgets(zeile,100,ifil) == NULL) break;
  /* Ignorieren von Leerzeilen (nur Linefeed) */
  if (zeile[0] == 0x0a) continue;
  if (fputs(zeile,ofil) == EOF) break;
}
```

Im Unterschied zur *while*-Schleife wird die *do while*-Schleife immer mindestens einmal durchlaufen, da die Abfrage der Endebedingung erst nach Durchlaufen des Schleifenkörpers ausgeführt wird. Dieser Schleifentyp wird deshalb nur selten verwendet. Bitte denken Sie daran, daß die *do while*-Schleife zwei Schlüsselwörter benötigt. *do* steht vor dem Schleifenkörper, *while* dahinter. Beispiele:

```
/* Lesen aus einem File bis EOF erreicht wird */
char *hadr;
  do
  {
    /* fgets() liefert NULL im EOF - Fall */
    hadr = fgets(zeile,lzeile,pdatei);
  }  while (hadr != NULL);
```

Die *for*-Schleife sieht vordergründig so aus, als ob sie so funktionieren würde wie in den anderen Programmiersprachen. Bitte lassen Sie sich nicht täuschen. Sie ist um einiges mächtiger als etwa in Pascal oder BASIC. Sie hat die allgemeine Form:

```
for ([Initialisierung]; [Schleifenbedingung];
     [Aenderungssequenz])
   [Schleifenkoerper];
```

Der nach dem Schlüsselwort *for* erscheinende Klammerausdruck wird im folgenden Text auch Schleifenliste genannt. Mehrere Dinge sind für Umsteiger aus anderen Programmiersprachen ungewöhnlich. Initialisierung, Schleifenbedingung und Änderungssequenz bestehen aus beliebigen C-Ausdrücken. Hier kann der Kommaoperator dazu verwendet werden, mehrere Ausdrücke eines Elements der Schleifenliste voneinander zu trennen und die Abarbeitung in Links-Rechts Reihenfolge zu erzwingen. Ungewöhnlich ist auch, daß es keine Festlegung der Bedeutung der Ausdrücke in der Schleifenliste gibt. Dort müssen also nicht unbedingt Laufvariablen verwendet werden. Es gibt Programmierkünstler, die Befehle, die in anderen Programmiersprachen im Schleifenkörper erscheinen würden, komplett in der Schleifenliste unterbringen. Dies fördert allerdings kaum die Lesbarkeit von C-Programmen. C ist hier wieder sehr liberal – und fehlerträchtig!

Die in Klammern stehende Schleifenliste **muß** drei C-Ausdrücke umfassen, die durch Semikolons getrennt werden. Einzelne oder alle Ausdrücke der Schleifenliste können allerdings leer sein. Das trennende Semikolon muß dann trotzdem erscheinen. Die *for*-Schleife wird wie die übrigen Schleifen dann abgebrochen, wenn die Auswertung der Schleifenbedingung den Wert Null ergibt. Die Befehle *break* und *continue* können

mit der gleichen Bedeutung wie in der *while*-Schleife benutzt werden. Einige Beispiele für *for*-Schleifen:

```
/* Eine Endlosschleife mit leerem Schleifenkoerper */
for (;;) ;

/* Eine Endlosschleife mit einfachem Befehl als  */
/* Schleifenkoerper                              */
for (;;)
  if (fgets(zeile,100,ifil) == NULL) break;

/* Endlosschleife mit zusammengesetztem Befehl als */
/* Schleifenkoerper                                */
/* Ausstieg ueber break                            */
for (;;)
{
  hadr = fgets(zeile,100,ifil);
  if (hadr == NULL) break;
  fputs(zeile,ofil);
}

/* Schleife a la PASCAL und FORTRAN */
for (n = 0; n < 10; n++)
  a = b + array[n];

/* Schleife mit quadratischem Wachstum der Laufvariablen */
for (n = 2; n < 1024; n = n * n)
  printf("\nn= %d",n);

/* Schleife ohne Schleifenvariable im klass. Sinn  */
/* statt "p;" koennte auch "p != NULL;" erscheinen */
for (p = startpointer; p; p = p->next)
  printf("\nnaechstes Element: %s",p->element);

/* Schleife mit Kommaoperator in Initialisierung und    */
/* Aenderungssequenz                                    */
/* Hier wird fast alle Arbeit in der Schleifenliste     */
/* erledigt. Ergebnis: cfeld=ABCDEFGHIJ                 */
char cfeld[10],c[2];
int n;
for
  (n = 0, cfeld[n] = 0, c[0] = 'A', c[1] = 0; /* Initial.  */
  n < 10;                                     /* Bedingung */
  strcat(cfeld,c), c[0] += 1, n++)            /* Aenderung */
    printf("\n = %d, c[0] = %c",n,c[0]);      /* Koerper   */
```

Das letzte Beispiel verwendet den **Kommaoperator** im Initialisierungsausdruck und in der Änderungssequenz. Der Kommaoperator sorgt dafür, daß die Teilausdrücke (die Operanden des Kommaoperators) in Links-Rechts-Reihenfolge abgearbeitet werden.

Die Reihenfolge der Bearbeitung der Ausdrücke in der Schleifenliste einer *for*-Schleife wird deutlich, wenn ich die *for*-Schleife als *while*-Schleife formuliere. Ich formuliere dazu das letzte Beispiel um:

```
/* Uebersetzung einer for in eine while-Schleife */
/* Initialisierung */
n        = 0;
cfeld[n] = 0;
c[0]     = 'A';
c[1]     = 0;
/* Test der Schleifenbedingung */
while (n < 10)
{
  /* Schleifenkoerper */
  printf("\n = %d c[0] = %c",n,c[0]);
  /* Aenderungssequenz */
  strcat(cfeld,c);
  c[0] += 1;
  n++;
}   /* while n < 10 */
```

Die Reihenfolge der Abarbeitung sieht also so aus: Initialisierung, Test der Schleifenbedingung, Abarbeitung Schleifenkörper, Ausführen der Änderungssequenz, Rücksprung zum Test der Schleifenbedingung.

6.7 Der Sprungbefehl

Der Sprungbefehl *goto* ist in C mit Absicht etwas dürftig ausgefallen. Als C konzipiert wurde, war die Diskussion um die strukturierte Programmierung in vollem Schwange, und einer deren Glaubenssätze heißt ja: *goto* vermeiden. Da es aber dennoch sinnvolle Anwendungen für *goto* gibt, sieht C den Sprungbefehl vor. Sie sollten *goto* nur verwenden, um möglichst schnell aus einer komplizierten Blockschachtelung (Schachtelungen von *if*-Blöcken und Schleifen) herauszuspringen. Echte Schleifen können Sie in C immer mit den schon besprochenen Schleifenkonstrukten (also ohne *goto*) formulieren; in Normalfällen ersetzen die Befehle *continue* und *break* auch den Einsatz von *goto*, wenn eine Schleife nicht komplett durchlaufen oder verlassen werden soll.

Mit dem *goto*-Befehl ist der Begriff der **Sprungmarke** (label) verbunden. Eine Sprungmarke besteht aus einem Namen und einem Doppelpunkt. Der Name der Sprungmarke unterliegt den Namensregeln, die ich für die Wahl von Variablennamen angegeben habe. Die Sprungmarke hat in C keine andere Funktion, als das Sprungziel anzugeben. Obwohl inhaltlich die Sprungmarke eine Maschinenadresse ist, wird sie in C nicht wie eine Pointerkonstante behandelt. Sie kann also nicht einer Pointervariablen zugewiesen werden oder als Argument in einer Prozedur auftreten. Sprünge sind in C lokal zur jeweiligen Prozedur: mit *goto* können Sie also nicht aus einer Prozedur herausspringen. Ein sinnvolles Beispiel für *goto* (aus *dinsort.c* in Kap. 17):

```
for (n=0; n < 256; n++)
{
  if (hash[n] == NULL) continue;
  pchain = hash[n];
  while (pchain != NULL)
  {
    if (fprintf(ofil,"%s",pchain->orecord) < 0)
    {
      printf("\n--- Schreibfehler ---\n");
      goto ende;
    }
    printf("%s",pchain->orecord);
    pchain = pchain->next;
  }   /* while pchain != NULL */
}   /* for n=0; n <256; n++ */
ende:
  fclose(ifil);
  fclose(ofil);
```

Natürlich könnte ich hier auch mit *break* die *while*-Schleife verlassen. Dann jedoch würde die *for*-Schleife fortgesetzt, was ich jedoch vermeiden will, da ein Schreibfehler aufgetreten ist. Das Problem liegt darin, daß mit *break* nur die aktuelle Schleife verlassen werden kann; ich will jedoch auch die *for*-Schleife verlassen. Die mögliche Lösung über *break* benötigt ein zusätzliches Flag oder das Setzen der Laufvariablen n auf einen Wert >= 256. Das jedoch ist auch kein feiner Weg.

6.8 Zusammenfassung

Folgende Befehlstypen müssen unterschieden werden:

- Präprozessorbefehle
- C-Befehle

Die Präprozessorbefehle sind keine Befehle im Sinne der C-Sprachdefinition. Die C-Befehle lassen sich nach verschiedenen Kriterien unterteilen in:

- Befehle ohne Schlüsselwörter (Ausdrücke)
- Befehle mit Schlüsselwörtern
- ausführbare Befehle
- nicht ausführbare Befehle (Deklarationen)
- einfache Befehle
- zusammengesetzte Befehle (Blöcke)

Schlüsselwörter werden in C kleingeschrieben. Einfache Befehle enden mit einem Semikolon, zusammengesetzte Befehle mit einer schließenden geschweiften Klammer.

Hier die Liste der C-Schlüsselwörter, die in ausführbaren Befehlen erscheinen können:

```
if          continue
do          goto
while       switch
for         break
case        default
return
```

7 Ein- und Ausgabe

7.1 Die Rolle der C-Bibliothek

Die Sprache C hat keine Befehle zur Ein- oder Ausgabe von Daten. Alle Eingaben und Ausgaben werden über Funktionen der C-Bibliothek abgewickelt, die ich in einem eigenen Kapitel behandle. Wenn Sie also auf eine C-Implementierung treffen, die die eine oder andere Bibliotheksroutine zur Ein-/Ausgabe nicht enthält oder ein leicht andersartiges Verhalten aufweist, schieben Sie die Schuld nicht auf C. Es war Absicht der Entwickler von C, die Ein-/Ausgabe in die C-Bibliothek zu stecken, da gerade die Ein-/Ausgabe stark von der jeweiligen Maschine und vom Betriebssystem abhängt. Die C-Bibliothek aber muß beim Erstellen eines Compilers immer an die jeweilige Umgebung angepaßt werden. Die Ein-/Ausgabe ist dort also gut aufgehoben. Funktionen, die in diesem Kapitel nicht oder nur am Rande behandelt werden, sollten Sie im Kapitel über die C-Bibliothek erläutert finden, wenn sie nicht allzu speziell sind. Ich begnüge mich hier damit, die Philosophie der Ein-/Ausgabe zu erkären.

Angeblich gibt es eine 'C-Standardbibliothek' – zumindest heißt sie so. (Eine der wichtigsten Aufgaben der kommenden ANSI-Norm für C besteht darin, tatsächlich eine Standardbibliothek zu definieren). Alle Anbieter von C-Compilern versuchen, wenigstens die wichtigsten Funktionen aus K&R bereitzustellen, so daß auch auf dieser Ebene (mit einigen Einschränkungen) portabel programmiert werden kann. Wenn Sie PASCAL kennen, wissen Sie vielleicht, daß gerade die (dort in der Sprache definierte) Ein-/Ausgabe derart unzureichend ist, daß einige Anwender (ich gehörte auch dazu) niemals die angebotenen Sprachmittel benutzt haben.

7.2 UNIX-Philosophie der Ein-/Ausgabe

Die Ein-/Ausgabefunktionen der C-Bibliothek werden nur dann verständlich, wenn die Philosophie der Ein-/Ausgabe im Betriebssystem UNIX bekannt ist. Sie erinnern sich: C wurde als Systemprogrammierungs- und Anwendungssprache für UNIX entwickelt. Die in K&R definierten Ein-/Ausgabefunktionen beziehen sich auf UNIX. Die Ein-/Ausgabefunktionen in K&R auf anderen Maschinen nachbilden heißt deshalb letztlich, UNIX-Eigenschaften nachzubilden. Kein Wunder, daß dies manchmal schief geht. Man kann nicht erwarten, daß ein Betriebssystem XYZ die gleichen Eigenschaften wie UNIX aufweist. Bei einigen Betriebssystemen wie MS-DOS und Amiga-DOS finden sich allerdings soviele Parallelen zu UNIX, daß sich mir die Frage stellt: wer hat hier von wem gelernt?

Die wichtigsten Leitsätze der UNIX Ein-/Ausgabephilosophie lassen sich so umreißen:

- baumartiges Dateisystem
- einheitliche Behandlung von Dateien und Ein-/Ausgabegeräten
- wenige Basisfunktionen für die Ein-/Ausgabe: höhere Funktionen bauen auf den Basisfunktionen auf.
- eingebautes System der Zugriffsrechte und der Besitzrechte.

UNIX verfügt über ein hierarchisches, baumartig organisiertes Dateisystem. Jedem Knoten im Baum entspricht eine Datei. Die Endknoten heißen Dateien (Files); Nicht-Endknoten heißen Verzeichnis (Katalog, Directory). Dateinamen sind infolgedessen Pfadnamen, d.h. sie geben den Weg vom Wurzelknoten (der 'root') eines Dateisystems bis zur angesprochenen Datei an. In MS-DOS, TOS und AmigaDOS finden Sie ähnliche Konzepte. In CP/M finden Sie eher ein 'flaches' Dateisystem vor; es gibt dort keine Unterverzeichnisse (sieht man vom 'User'-Konzept ab). In CP/M-orientierten Compilern werden deshalb wohl keine Pfadnamen wie *c:\cbuch\io* als gültige Pfadnamen akzeptiert. Der Amiga kennt dieses Problem allerdings nicht!

7.3 Familien von Ein-/Ausgabefunktionen

Die wichtigsten Ein-/Ausgabeprozeduren der C-Bibliothek lassen sich in zwei Familien ordnen:

- Basisprozeduren: sie werden auch low-level Routinen (Routinen der unteren Ebene) oder Block-IO-Routinen genannt. Treffender finde ich den Ausdruck UNIX-IO-Routinen, da diese Prozeduren die untersten und elementarsten Routinen für Ein-/Ausgabe in UNIX darstellen. Darunter liegen in UNIX nur noch die Gerätetreiber. Zu den Basisprozeduren gehören *read(), write, open, close()* etc.

- Standardprozeduren: sie werden auch high-level Routinen (Routinen der oberen Ebene) genannt. Zu Ihnen gehören die schon bekannten Routinen wie *fopen(), fclose() , gets(), fgets(), printf(), getchar()* etc.

Die beiden Familien lassen sich nach weiteren Merkmalen klassifizieren. Die Basisroutinen sind immer dateiorientiert (auch Tastatur und Bildschirm werden in UNIX bzw. C wie Dateien behandelt). Dateien im Sinne der Basisprozeduren werden über Nummern (wie in Fortran oder BASIC) angesprochen. Die Standardprozeduren sparen sich die Dateiangaben, wenn es um Eingabe oder Ausgabe zum Terminal geht. Die Standardroutinen ohne Dateiangabe (z.B. *gets(), printf(), puts(), getchar(), putchar()*) weichen jedoch nur vordergründig von den dateibezogenen Standardroutinen (z.B. *fgets(), fprintf(), fputs(), fgetc(), fputc()*) ab. In Wirklichkeit sind die Terminalroutinen meist nur Kurzformen (Makros) der dateiorientierten Varianten. Im Kapitel über die C-Bibliothek werden die Verwandtschaftsverhältnisse bei den jeweiligen Prozeduren erörtert. Jetzt aber das wichtigste: die Standardroutinen greifen nicht über Nummern auf Dateien zu, sondern über **Filepointer**. Filepointer sind Pointer auf Strukturen des Typs FILE. Dieser Typ ist über *typedef* oder über einen Präprozessorbefehl im *#include*-File *stdio.h* deklariert. Damit wird auch klar, weshalb in nahezu allen C-Programmen der Name dieses *#include*-Files erscheint. In *stdio.h* werden noch weitere Objekte (Makros, Konstanten wie *EOF*) definiert, die für die Standardprozeduren zur Ein-/Ausgabe erforderlich sind. Im Kapitel über den C-Präprozessor ist ein Ausschnitt aus einer Version von *stdio.h* abgedruckt.

Die Basisroutinen erlauben einen systemnahen, sehr effizienten Umgang mit Dateien. Dateien werden dabei einfach als Folgen von Bytes betrachtet, die in größeren Portionen gelesen werden. Die Basisroutinen in ihrer K&R Version nehmen keine Interpretation der gelesenen Daten vor; sie sind also für Binärdateien geeignet und damit transparent. Leider sind inzwischen viele C-Bibliotheken von diesem Grundsatz abgerückt. Die Standardroutinen bauen auf den Basisroutinen auf, d.h. irgendwo (für den Programmierer unsichtbar) werden Routinen wie *read()* oder *write()* aufgerufen, wenn mit *fgets()* oder *fputs()* gearbeitet wird. Die Standardroutinen erlauben eine eher logisch orientierte Sicht von Dateien: Dateien sind demnach Folgen von Zeichen, von Zeilen oder gar von formatierten Objekten wie Zahlen und Strings. Konsequenterweise liefern die Standardroutinen nicht alle Zeichen, die in einem File erscheinen. So liefern z.B. einige Versionen von *fgetc()* kein Carriage Return (0x0d), wenn es hinter einem Linefeed (0x0a) in einem File erscheint. Ein Zeilenendzeichen ist wohl genug. *fputc()* ergänzt auf vielen Maschinen das Zeichen Linefeed zu Linefeed + Carriage Return. Die Prozedur *gets()* liefert überhaupt kein getastetes Zeilenendzeichen, sondern statt dessen eine binäre Null (das Stringendezeichen).

In C müssen Dateien zu Beginn der Verarbeitung immer geöffnet werden und am Ende der Verarbeitung geschlossen werden. Warum muß eine Datei geöffnet werden? Dateien werden in C entweder über Nummern oder über Filepointer angesprochen. Die Prozeduren zum Öffnen einer Datei liefern diese Nummern bzw. Filepoin-

ter. Im Falle der Standardroutinen werden beim Öffnen eines Files interne Puffer allokiert. In C gibt es mindestens drei Nummern und drei Filepointer, deren entsprechende Files immer offen sind und auch nicht geschlossen werden müssen. Diese Files sind die wichtigsten Ein-/Ausgabewege für den Terminalbetrieb. Zur Unterscheidung von Dateien im landläufigen Sinn wird hier auch von Kanälen gesprochen:

Basisroutinen	Standardroutinen
0	stdin
1	stdout
2	stderr

0 bzw. *stdin* steht für die Standardeingabe (Tastatur), 1 bzw *stdout* für die Standardausgabe (Bildschirm) und 2 bzw *stderr* steht für die Standardfehlerausgabe (Bildschirm). 0, 1 und 2 sind Konstanten; In C gibt es keine Variablen mit Standardnamen oder Konstantensymbole für diese Kanäle in den Basisroutinen. *stdin, stdout* und *stderr* jedoch sind in *stdio.h* definiert. Sie brauchen sie also nicht extra zu deklarieren. Die Standardkanäle der Basisroutinen und der Standardroutinen werden zwar unterschiedlich angesprochen, sind aber physikalisch gleich! Die Standardroutinen greifen z.B. über *stderr* auf den Kanal 2 zu. Deshalb sollten Sie es unbedingt vermeiden, Basisroutinen und Standardroutinen für die gleiche Datei oder für das gleiche Gerät zu mischen. Eine solche Mischung führt mit einiger Wahrscheinlichkeit zu Fehlern.

Warum ein zweiter Ausgabekanal, nämlich 2 bzw. *stderr* nötig ist? Im Kapitel 7.4 werde ich über die Umlenkung der Ein-/Ausgabe sprechen. Wenn die Standardausgabe (1 bzw *stdout*) in eine echte Datei umgelenkt wird, verschwinden auch die Fehlermeldungen. Ein gewitzter Programmierer (wie Sie an den Beispielsprogrammen sehen, bin ich nicht immer so gewitzt) wird deshalb Fehlermeldungen nie über den Standardausgabekanal ausgeben, sondern über den Kanal 2 bzw. *stderr*. Dann erscheinen die Fehlermeldungen am Schirm, selbst wenn die Standardausgabe umgelenkt wurde!

Ich gebe jetzt eine Auflistung der wichtigsten Funktionen zur Ein-/Ausgabe nach logischen Gesichtspunkten (was macht die Funktion?). Details zu den einzelnen Funktionen finden Sie in Kapitel 12. Zunächst die Basisroutinen:

logische Funktion	Name der Prozedur
Anlegen	creat()
Oeffnen	open()
Schliessen	close()
Lesen	read()
Schreiben	write()
Positionieren	lseek()

Jetzt die Standardroutinen. Die logischen Funktionen sind vielfältiger. Eine Funktion, die *creat()* entspricht, gibt es nicht in den Standardroutinen. Die Funktion *fopen()* übernimmt diese Aufgabe mit. Ich gebe nur die dateiorientierten Routinen an. Die terminalorientierten Routinen sind – bis auf kleine Abweichungen – nur Varianten der dateiorientierten Routinen. Details können wieder dem Kapitel 12 entnommen werden:

log. Funktion	Name der Funktion
Oeffnen	fopen()
Schliessen	fclose()
Lesen zeilenweise	fgets()
Lesen zeichenweise	fgetc()
Schreiben zeilenweise	fputs()
Schreiben zeichenweise	fputc()
Lesen formatiert	fscanf()
Schreiben formatiert	fprintf()
Positionieren	fseek()

Ich habe in den vorhergehenden Beispielsprogrammen ständig terminalorientierte Prozeduren wie *printf()* und *gets()* verwendet. Hier dennoch ein kleines Beispielsprogramm, das Sie dringend testen sollten, um das Verhalten der terminalorientierten Standardprozeduren an Ihrer Maschine zu erfahren. Das Programm liest einfach nur Zeichen, Zeilen und formatierte Objekte (Dezimalzahlen) ein und gibt sie wieder aus.

Sie sollten die Eingaben für das nachstehende Programm variieren und absichtlich falsche Eingaben liefern, um einen umfassenden Eindruck vom Verhalten dieser zentralen Prozeduren zu bekommen.

```
/****************************************

          Terminal-Testprogramm
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Testet die terminalorientierten Standartprozeduren

****************************************/

#include <stdio.h>

main()
{
  int ic,ivar1,ivar2;
  char cfeld[100],*hadr;

  /* Test getchar - putchar */
  printf("\nZeichenweise Ein/Ausgabe");
  while (1)
    {
     printf("\nZeichen+CR:");
     ic = getchar();
     if (ic == EOF) break;
     puts("\ngelesen:");
     putchar(ic);
    }
  printf("\n--- EOF erkannt ---");

  /* Test gets - puts */
  printf("\nZeilenweise Eingabe:");
  while (1)
    {
     printf("\nZeile+CR:");
     hadr = gets(cfeld);
     if (hadr == NULL) break;
     printf("\ngelesen ");
     puts(cfeld);
    }

  /* Test printf - scanf */
  printf("\nformatierte Zahleneingabe:");
  while (1)
    {
     puts("\nZwei Dez.Zahlen:");
     ic = scanf("%d %d",&ivar1,&ivar2);
     if (ic < 2) break;
     printf("\nivar1= %d ivar2= %d",ivar1,ivar2);
    }
  printf("\n--- Programmende ---");
}   /* end main */
```

Der Gebrauch der wichtigsten fileorientierten Ein-/Ausgaberoutinen soll an den folgenden Beispielen gezeigt werden. Es geht jedesmal um das Anlegen der Kopie eines Files. Die Variante 1 verwendet die zeichenorientierten Standardroutinen. Die Variante 2 verwendet die zeilenorientierten Standardroutinen. Die Standardroutinen können bisweilen auch für den Binärbetrieb verwendet werden. Das ist jedoch sicher nicht transportabel. Bitte achten Sie bei den Beispielen insbesondere auf die Rückgabewerte der verwendeten Funktionen. Die Basisfunktionen liefern fast immer *int*-Werte (Ausnahme *lseek:* liefert *long*-Wert), die Standardfunktionen manchmal Pointer und manchmal *int*-Werte! Für den Benutzerdialog werden immer die Standardroutinen verwendet. Die Basisroutinen wären hier zu unhandlich.

```
/****************************************

      .  File-Kopierer Variante 1
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Filekopie: Variante 1: Verwendung des Standard IO
Lesen und Schreiben erfolgt zeichenweise
(Nicht fuer Binaerfiles geeignet)

****************************************/

#include <stdio.h>

abbruch(str)
char *str;
{
  fprintf(stderr,"\n--- %s ---\n",str);
  exit(1);
}   /* end abbruch */

main()
{
  char filna[60];
  FILE *ifil,*ofil;
  int ic;

  fprintf(stderr,"\nInputfile:");
  gets(filna);
  ifil = fopen(filna,"r");
  if (ifil == NULL) abbruch("Fehler Inputfile");
  fprintf(stderr,"\nName der Kopie:");
  gets(filna);
  ofil = fopen(filna,"w");
  if (ofil == NULL) abbruch("Fehler Ausgabefile");
```

```
  while (1)
    {
     ic = fgetc(ifil);
     if (ic == EOF) break;
     ic = fputc(ic,ofil);
     if (ic == EOF) abbruch("Schreibfehler");
    }

  fclose(ifil);
  fclose(ofil);
}   /* end main */


/****************************************

         File-Kopierer Variante 2
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Filekopie: Variante 2: Verwendung des Standard IO
Lesen und Schreiben erfolgt zeilenweise
(Nicht fuer Binaerfiles geeignet)

****************************************/

#include <stdio.h>

abbruch(str)
char *str;
{
  fprintf(stderr,"\n--- %s ---\n",str);
  exit(1);
}   /* end abbruch */

main()
{
  char filna[60],zeile[100],*hadr;
  FILE *ifil,*ofil;
  int ic;

  fprintf(stderr,"\nInputfile:");
  gets(filna);
  ifil = fopen(filna,"r");
  if (ifil == NULL) abbruch("Fehler Inputfile");
  fprintf(stderr,"\nName der Kopie:");
  gets(filna);
  ofil = fopen(filna,"w");
  if (ofil == NULL) abbruch("Fehler Ausgabefile");

  while (1)
    {
     hadr = fgets(zeile,100,ifil);
     if (hadr == NULL) break;
```

```
      ic = fputs(zeile,ofil);
      if (ic == EOF) abbruch("Schreibfehler");
    }

  fclose(ifil);
  fclose(ofil);
}   /* end main */
```

7.4 Umlenkung der Ein-/Ausgabe

Von 'Umlenkung' (redirection) der Ein-/Ausgabe spricht man, wenn statt der Eingabe von Tastatur die Eingabe aus einem File erfolgt oder wenn die Ausgabe auf einen Schirm durch die Ausgabe in eine Datei ersetzt wird. Wichtig dabei ist: am Programm braucht nichts geändert zu werden, um diese Umlenkung zu erreichen.

Wie ist das möglich? Ganz einfach durch die Befolgung der UNIX-Philosophie, daß Dateien (Files) und Geräte zur Ein-/Ausgabe (Schirm, Tastatur, serielle Schnittstelle etc.) prinzipiell gleichbehandelt werden. Von der Ebene des Anwendungsprogramms sollte es prinzipiell gleichgültig sein, ob ein Gerät oder eine Datei angesprochen wird. Natürlich ist diese Philosophie nicht in allen Betriebssystemen problemlos anwendbar. Sie sollten also auch hier mit Schwierigkeiten rechnen, wenn Sie nicht unter UNIX arbeiten.

Bei der Umlenkung der Ein-/Ausgabe sind zwei Fälle zu unterscheiden: Die Umlenkung kann innerhalb des C-Programms erfolgen oder aber vom Kommandointerpreter veranlaßt werden.

Die erste Variante – Umlenkung innerhalb eines C-Programms – ist in allen Betriebssystemen durchführbar. Sie erfordert lediglich etwas Flexibilität beim Design des Programms und die Übergabe einer Option (eines Arguments) an *main(),* das den Wunsch nach Umlenkung signalisiert. Das folgende Beispiel zeigt die Umlenkung der Ausgabe in eine Datei, falls in der Kommandozeile die Option *-o* gefolgt von einem Dateinamen angegeben wird. Der Mechanismus der Übergabe von Argumenten an *main()* wird eigentlich erst im nächsten Abschnitt beschrieben. Wichtig ist hier eigentlich nur die Umsetzung des Filepointers *fpoi* von *stdout* auf die angegebene Datei:

```
/****************************************

            Ausgabe-Umleitung
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Umleitung der Ausgabe mit C-Befehlen
Aufruf: redirect [-o datei]

****************************************/

#include <stdio.h>

main(argc,argv)
int argc;
char *argv[];
{
  FILE *fpoi;
  char zeile[80];

  /* Vorbesetzung fuer fpoi auf Standardausgabe */
  fpoi = stdout;
  if (argc > 2)
    {
     if (strcmp(argv[1],"-o") == 0)
       {
        /* Option -o dateiname auswerten */
        /* fpoi wird umgesetzt           */
        fpoi = fopen(argv[2],"w");
        if (fpoi == NULL) exit(1);
       }
    }
  /* Zeile vom Terminal lesen und auf Schirm */
  /* oder in die angegebene Datei ausgeben   */
  /* Ende der Eingabe: CTRL-Z (MSDOS!)       */
  while (1)
    {
     if (fgets(zeile,80,stdin) == NULL) break;
     fprintf(fpoi,"%s",zeile);
    }

  if (fpoi != stdout) fclose(fpoi);
}   /* end main */
```

Das Programm liest Zeilen vom Terminal ein und schreibt sie entweder wieder auf das Terminal oder in die hinter *-o* angegebene Datei. Der Wunsch nach Umlenkung der Ausgabe wird hier durch die Option *-o* plus Dateiname ausgedrückt. Die nötige Programmierflexibilität wird durch die Wahl der fileorientierten Routinen *fopen()* und *fprintf()* erreicht sowie durch die Wahl einer Filevariablen *fpoi* statt der Filekonstanten *stdout*. Aus dem dummen Programm zum Auflisten des Eingelesenen wird so einfach ein etwas weniger dummes Programm zum Texterfassen.

Das obige Beispiel läßt sich leicht um doppelte Umlenkung (Umlenkung der Eingabe und der Ausgabe) erweitern. Nach dem gleichen Prinzip läßt sich auch Umlenkung der Ein-/Ausgabe auf irgendwelche Schnittstellen erreichen. Natürlich können Sie auch die UNIX-IO-Routinen (*read, write, open, close*) statt der Standard-IO-Routinen verwenden.

Die zweite angesprochene Möglichkeit – Umlenkung der Ein-/Ausgabe durch den Kommandointerpreter – ist sogar noch einfacher. Das gerade aufgelistete Beispielsprogramm braucht in diesem Fall weder die fileorientierten Routinen zu verwenden noch die Argumente an *main*() auszuwerten: es kann so geschrieben werden, als ob es immer nur mit dem Terminal zu tun hätte. Der Aufruf sieht im Falle der Ausgabeumlenkung so aus:

```
redirect >dateiname
```

und im Falle der doppelten Umlenkung:

```
redirect <dateiname1 >dateiname2
```

Die Umlenkung gilt dann übrigens nur für die Dauer des Programmlaufes; nach Programmende gelten wieder die üblichen Einstellungen: *stdin* bezieht sich auf die Tastatur, *stdout* auf den Bildschirm.

7.5 Prozeßkommunikation

UNIX ist ein Multitasking-Multiuser-System. Jeder Benutzer kann gleichzeitig mehrere Prozesse laufen haben: der Kommandoprozessor ist z.B. solch ein Prozeß, der Editor, der Compiler, die eigene Anwendung, das aktuelle Kommando. Der Kommandoprozessor (in UNIX-Terminologie die 'Shell') erzeugt beim Starten eines Kommandos einen Sohnprozeß, auf dessen Ende er wartet. Sohnprozesse erben vom Vaterprozeß die gemeinsame Prozeßumgebung, z.B. systemweite und prozeßweite Variablen.

Wenn mehrere Prozesse laufen, müssen sie sich untereinander verständigen können. Dazu gibt es in UNIX mehrere Möglichkeiten: die Übergabe von Argumenten an einen Prozeß, das Abfragen von Rückgabewerten eines Prozesses, pipes (gemeinsame Kommunikationskanäle), Austausch von Signalen (Semaphoren). Wie Sie sehen, werden Prozesse in UNIX fast wie Unterprogramme behandelt – mit dem Unterschied, daß sie parallel zueinander ablaufen können und bei Bedarf allein und parallel lebensfähig sind. Ein Unterprogramm ist ja nicht allein lebensfähig und läuft auch nicht parallel zum rufenden Programm ab.

Von diesen UNIX-Mitteln finden Sie mit einiger Sicherheit nur zwei in allen C-Compilern: die Übergabe von Argumenten und das Abfragen des Rückgabewertes eines Prozesses. Im Amiga-Betriebssystem sind allerdings noch andere Kommunikationsmöglichkeiten vorhanden.

Die Übergabe von Argumenten zwischen Prozessen beschränkt sich auf die Prozesse Kommandoprozessor (Shell) und Ihre jeweilige Anwendung (das gelinkte und ausgeführte Programm). An Ihr Hauptprogramm *main()* können Sie die Argumente der Kommandozeile übergeben. In *main()* können Sie diese Argumente auswerten und umsetzen, z.B. in Filenamen, in Testoptionen etc. Das Programm *dinsort.c* in Kapitel 17 kann z.B. mit der Option *-t* versehen werden und erzeugt dann einen ausführlicheren Ausdruck, der bei der Fehlersuche hilfreich ist. Das folgende Beispielprogramm druckt nur die übergebenen Argumente aus:

```
/****************************************

          Argumentuebergabe-Test
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Testet die Argumentenuebergabe an main()

****************************************/

#include <stdio.h>

main(azaehler,aadressen)
int azaehler;
char *aadressen[];
{
  int n;

  printf("\nAnzahl Argumente: %d",azaehler);
  for (n = 0; n < azaehler; n++)
    printf("\nArgument Nr. %d  %s",n,aadressen[n]);
}   /* end main */
```

```
Ergebnis fuer: mainarg 10 vater -t

  Anzahl Argumente: 4
  Argument Nr. 0
  Argument Nr. 1  10
  Argument Nr. 2  vater
  Argument Nr. 3  -t
```

Falls die übergebenen Argumente ausgewertet werden sollen, muß *main()* zwei Argumente ausweisen: eine Variable vom Typ *int* (hier *azaehler*), in der die Zahl der übergebenen Argumente abgelegt werden und ein Array von Pointern des Typs (*char* *) (hier *aadresse[]*), in dem die Adressen der Argumente abgelegt werden. Meist werden die Namen *argc* und *argv* gewählt. Ich wollte hier zeigen, daß auch andere Namen möglich sind. Wie Sie an der Ausgabe des Beispielsprogramms sehen, zählt auch der Kommandoname als Argument. Zwei kleine Komplikationen sind mir in

der Praxis begegnet: die Auswertung des Kommandonamens in *aadresse[0]* bringt nicht immer sinnvolle Resultate (s. obiges Ergebnis). Befehle zur Umleitung der Ein-/Ausgabe (*>file* oder *<file*) werden nicht als Argumente gezählt und erscheinen nicht im Array der Argumentpointer! Diese Art der Argumentübergabe funktioniert so nur in kommandoorientierten Betriebssystemen wie dem CLI.

Vor allem in Batchprozeduren ist es sinnvoll, den Rückgabewert eines Prozesses abzufragen. Falls ein Prozeß (ein Kommando) mit einem Fehler beendet wurde, kann es z.B. sinnvoll sein, die komplette Batchprozedur abzubrechen. Deshalb kann man in C ein Programm mit *exit(returnwert)* verlassen; *exit(returnwert)* kann an beliebiger Stelle in einem Programm stehen. Es muß also nicht am Ende von *main()* erscheinen. In UNIX ist es üblich, den Wert 0 an den Vaterprozeß (meist den Kommandoprozessor) zu liefern, wenn alles korrekt abgelaufen ist und sonst einen Wert ungleich Null.

exit() ist eine Funktion der C-Bibliothek. Falls kein *exit()* in Ihrem Programm erscheint, wird doch eine solche Funktion ausgeführt, und zwar bei der Rückkehr aus *main*().

8 Arrays und Pointer

Der deutsche Ausdruck für 'Array' ist 'Feld' und der für 'Pointer' ist 'Zeiger' oder 'Adresse'. Ich verwende weiter die englischen Bezeichnungen, weil sie in der deutschen C-Welt geläufiger sind als die deutschen Bezeichnungen.

Es erscheint vielleicht seltsam, diese beiden Konzepte in einem gemeinsamen Kapitel zu behandeln. Sie werden jedoch sehen, daß dies sinnvoll ist, denn Arrays können durch Pointer ersetzt werden. Wenn Sie Assembler können, wird sie das nicht wundern: dort gibt es keine Arrays im strengen Sinne, wohl aber Pointer.

In allen höheren Programmiersprachen gibt es Arrays; Pointer dagegen gibt es nur in den neueren höheren Programmiersprachen. Fortran, BASIC, Algol und Lisp kennen noch keine Pointer auf der Sprachebene. Mir sind Pointer erst in PL/1 aufgefallen – eine Sprache, die seit Mitte der 60er Jahre existiert und somit mindestens fünf Jahre jünger ist als Fortran, Lisp und Algol. PL/1 hatte ein einfaches, aber effizientes Pointer-Konzept. Pascal, das später entstand, stellt (nicht nur in dieser Hinsicht, da mögen die Informatiker noch so toben) einen Rückschritt gegenüber PL/1 dar, weil die Pointer den rigiden Typrestriktionen unterworfen sind.

C ist bekannt für sein flexibles Pointerkonzept. Da Arrays durch Pointer erklärt werden können, beginne ich mit Pointern.

8.1 Pointer

Pointer haben nichts Geheimnisvolles an sich: sie sind schlicht und einfach **Maschinenadressen**. Ich habe weiter oben schon erwähnt, daß Variablen mehrere Aspekte haben, von denen drei hier von besonderem Interesse sind: sie haben einen Wert, eine Adresse im Speicher und einen Namen. Viele Konstanten haben ebenfalls einen Wert (der unveränderlich ist) und eine Adresse im Speicher, auch wenn diese

Adresse (außer für Stringkonstanten) in C nicht zugänglich ist. Schließlich haben Programme auch eine Adresse im Speicher: die Adresse, die ihnen vom Linker (Binder) zugewiesen wurde. In den meisten Programmiersprachen sieht man diesen zweiten Aspekt – die Adreßseite – von Elementen einer Programmiersprache nicht. Tatsächlich lassen sich nahezu alle Probleme auch ohne Pointer formulieren – fragt sich nur, wie einfach und wie effizient.

Daß C Pointer als Sprachelement zuläßt, ist nichts Besonderes. Auch Pascal und PL/1 tun dies. In C allerdings ist mehr möglich: es gibt eine **Pointerarithmetik**, also ein Regelwerk, wie man auf Pointer die üblichen arithmetischen Operationen anwenden kann, also Addieren, Subtrahieren, Vergleichen etc. Dieser Aspekt der Pointerarithmetik ist entscheidend für die Stellung von C innerhalb der neueren höheren Programmiersprachen.

Pointer sind wie gesagt Maschinenadressen. Variablen – Sie erinnern sich – haben mindestens drei Aspekte: einen Variablennamen, einen Wert und eine Adresse im Speicher. Der Variablenname dient in C und in anderen höheren Programmiersprachen nur dazu, die Adresse vor dem Programmierer zu verbergen. Assemblerprogrammierer sind es gewöhnt, mit Adressen umzugehen, erst recht wenn sie systemnahe Programme schreiben – z.B. Treiber für Peripheriegeräte. Es gibt tatsächlich immer noch Gelegenheiten für Programmierer, mit echten Adressen zu hantieren. Je kommerzieller und anwendungsorientierter die Programmierung, um so weniger ist es nötig, Maschinenadressen in diesem Sinne zu kennen. Es gibt aber ein weiteres Argument für das Hantieren mit Pointern: sie machen einige Typen von Programmen einfacher und überschaulicher.

Normalerweise wird die Adreßberechnung vom Compiler, vom Linker und vom Lader erledigt – Sie wissen üblicherweise nicht, wo Ihr Programm abläuft. Je größer das System, um so weniger Informationen haben Sie über die Lage eines Programms und der darin enthaltenen Adressen. Bei kleinen Systemen kennt man einige Adressen vielleicht: in CP/M 2.2 beginnt die TPA (transient program area) auf Adresse 0x100, der wichtige BDOS-Call geht über die Adresse 0x05. Auch die Adresse 0x03 – das IO-Byte – ist eventuell bekannt. Und schließlich kennt man vielleicht noch den Begriff DMA im CP/M-Sinne: das ist die Adresse, an der ein gerade gelesener Disk-Sektor abgelegt wird. Wenn schon jemand einmal einen Treiber für Rechner der MC68000-Familie geschaut hat, wird er weitere feste Adressen gesehen haben: die Adressen von IO-Bausteinen, also von seriellen oder parallelen Schnittstellen, von Floppy-Controllern, von Sound-Chips etc. Es gibt also schon Notwendigkeiten, konstante Maschinenadressen zu kennen.

Die erwähnte DMA-Adresse in CP/M ist eigentlich eine variable Maschinenadresse. Es gibt in MS-DOS und in CP/M Möglichkeiten, die DMA nach eigenem Gusto zu setzen. Wenn ich z.B. vorhabe, die DMA in den Adreßbereich meines Programms zu verlegen, tue ich gut daran, die alte DMA zu retten. Dafür dieses kurze Beispielprogramm:

```
char *altedma;      /* Vereinbarung Variable */
char neuedma[128]; /* Vereinbarung Array */
char *liesdma();    /* Vereinbarung Funktionstyp */
{
      /* Am Programmanfang: */
      altedma = liesdma();
      setzedma(neuedma);
      /* Am Programmende */
      setzedma(altedma);
}
```

Es ist nicht wichtig, was die Funktionen *liesdma()* und *setzedma()* tun. Es geht hier nur darum, wie Pointer (Maschinenadressen) vereinbart und gespeichert werden.

Hier werden gleich drei Datentypen vereinbart, die mit dem Konzept Pointer zu tun haben: zuerst wird eine Pointervariable vom Typ *char** vereinbart. Dieser gleiche Typ taucht in der Deklaration der Funktion *liesdma()* nochmals auf. Der Stern * zeigt an, daß ein Pointer vorliegt. Daß eine Variable vorliegt, zeigt die Vereinbarung eines Namens: *altedma*. Nochmals: Der Typ der Variablen ist *char **, der Name ist *altedma* – nicht ***altedma!* Dann wird noch ein Array mit dem Namen *neuedma* vereinbart. Sie werden weiter unten sehen, daß Arraynamen auch Pointer – allerdings konstante Pointer – sind. Und schließlich wird vereinbart, daß die Funktion *liesdma()* einen Wert vom Typ *char** liefert. Schließlich enthält das Programm selbst zwei Pointeroperationen: eine Zuweisung des Werts der Funktion *liesdma()* an die Variable *altedma* und die Übergabe der Pointerwerte von *neuedma* und *altedma* als Argumente der Funktion *setzedma()*.

Sie sollten sich merken, daß Pointervariablen vereinbart werden müssen, daß sie mit Werten belegt werden können und daß Werte daraus gelesen werden können. Kurzum: sie verhalten sich wie normale Variablen. Warum aber gibt es überhaupt einen Grund, Ihnen einen Typ zuzuordnen, wenn sie eh normale Maschinenadressen sind? Schließlich sind Maschinenadressen – sieht man es etwas oberflächlich – die Adressen von Bytes im Speicher, und denen ist es egal, ob sie Zeichen, ganze Zahlen oder Fließkommazahlen enthalten.

Der Grund liegt darin, daß in C die Arithmetik mit Pointern typgebunden ist. Wenn ich die Adresse einer Zeichenvariable um eins erhöhe, will ich auf das rechts danebenliegende Zeichen. Erhöhe ich dagegen die Adresse einer *int*-Variablen, so will ich zwei Bytes weiter. Schließlich fasse ich die *int*-Variable als eine Einheit auf – mich interessiert normalerweise nicht, wieviele Bytes *int*-Variablen belegen.

Das folgende Programm soll zeigen, wo Sie landen, wenn Sie unterschiedlich 'getypte' Pointervariablen um eins erhöhen. Wie Sie sehen, wird nur der *printf()*-Aufruf variiert durch Einsetzen unterschiedlicher Cast-Operatoren:

```
/****************************************

            Zeigerbeeinflussung
            last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Erhoeht einen Zeiger

****************************************/

#include<stdio.h>

char *adr;

main()
{
  adr = 0x500;
  printf("\n%lx",(char *)adr+1);
  printf("\n%lx",(int *)adr+1);
  printf("\n%lx",(long *)adr+1);
  printf("\n%lx",(float *)adr+1);
  printf("\n%lx",(double *)adr+1);
} /* end main */
```

Ergebnis:

```
          501    /* char *     */
          502    /* int *      */
          504    /* long *     */
          504    /* float *    */
          508    /* double *   */
```

Wenn Sie einen Datentyp mit einem Stern * und einem gültigen C Namen kombinieren, dann deklarieren Sie einen Pointer. Beispiele für korrekte Deklarationen von Pointern:

```
/* Deklaration einfacher Pointervariablen */
char *pchar;
int *pint1,*pint2;
long *plong1,*plong2;
float *pfloat;
double *pdouble;

/* Deklaration eines Pointers auf einen Pointer */
char **ppchar;
```

```
/* Deklaration eines Arrays von Pointern */
/* vom Typ char *                        */
char *pchar[20];

/* Deklaration eines Pointers auf eine Struktur und */
/* eines Arrays von Pointern auf Strukturen         */
struct
{
  int ivar;
  float fvar;
} *pstruct,*pstr[10];

/* Deklaration eines Pointers auf eine Funktion, die */
/* int liefern soll                                  */
/* Die Schreibweise (*pfunct) ist noetig!            */
int (*pfunct)();

/* Deklaration einer Funktion, die einen Pointer */
/* liefert                                       */
extern long *lfunct();
```

Konstanten sollten immer über einen Cast-Operator zu Pointern gemacht werden. Das folgende Beispiel zeigt, wie in CP/M dem IOBYTE (darin wird die Zuordnung von logischen zu physikalischen Geräten notiert) ein Wert zugeordnet wird. Das IOBYTE hat die Adresse 3:

```
/* Zugriff auf das IOBYTE in CP/M */
*(unsigned char *)0x03 = 0x84;
```

Wenn Sie in diesem Beispiel statt (*unsigned char* *) den Cast-Operator (*int* *) verwenden, zerstören Sie auch den Inhalt von Adresse 4, weil *int*-Variablen mindestens zwei Bytes belegen. Auch bei Pointerkonstanten sollten Sie also vorsichtig mit dem Typ umgehen.

Neben der Deklaration und dem Cast-Operator ist der Adreßoperator & der wichtigste Weg, einen Pointer zu erzeugen. Der Sternoperator * macht aus einem Pointer eine Variable mit einfachem Datentyp. Das folgende Beispielsprogramm zeigt den Zusammenhang zwischen Pointervariablen, Adreßoperator und Sternoperator. Der Variablen *ivar* wird auf drei verschiedene Arten der Wert 17 zugewiesen. Natürlich sind die Lösungen 2 und 3 etwas an den Haaren herbeigezogen:

```
/* Varianten einer Zuweisung */
int ivar,*pivar;
  /* Loesung 1 */
  ivar = 17;
  /* Loesung 2 */
```

```
pivar  = &ivar;
*pivar = 17;
/* Loesung 3 */
*(&ivar) = 17;
```

In *stdio.h* wird normalerweise die Pointerkonstante *NULL* definiert; *NULL* bezeichnet den leeren Pointer. Dieser leere Pointer wird benötigt, um das Ende einer Verkettung anzuzeigen oder um anzudeuten, daß eine Pointervariable noch keinen sinnvollen Wert hat.

Verkettete Listen werden mit den Bibliotheksroutinen für die dynamische Speicherallokierung *malloc()* und *free()* angelegt. Ein Beispielsprogramm für die Verwendung von *NULL* und für verkettete Listen findet sich im Kapitel 9.

8.2 Eindimensionale Arrays

Arrays (Felder) sind Folgen von gleichartigen Variablen, die über einen gemeinsamen Namen und den Arrayauswahloperator [] ansprechbar sind. Das ist die übliche Definition, die dennoch für C etwas zu eng ist.

Tatsächlich können Sie in C auch 'höhere Objekte' wie Strukturen (darüber rede ich später) zu Arrays zusammenschließen. Erkenntlich ist ein Array in C nur an den eckigen Klammern, die hinter dem Arraynamen erscheinen. Die Schreibweise in Deklaration und Verwendung von Arrays ist gleich; es gibt keine speziellen Kennwörter für die Definition von Arrays wie in Pascal ('ARRAY OF...'). Der folgende Auszug aus einem Programm definiert einen eindimensionalen Array für ganzzahlige, vorzeichenbehaftete Variablen (oder schlicht: der Array ist vom Typ *int*) der Größe 10. Dann wird das dritte (!) Element dieses Arrays mit dem Wert 17 verglichen:

```
int feld[10];
   if (feld[2] == 17) .....
```

Eigentlich nichts Ungewöhnliches. Daß ein Array vom Typ *int* mit der Größe 10 vorliegt, wird aus der Deklarationszeile deutlich. Seltsam ist nur, daß mit dem Ausdruck *feld[2]* das dritte (nicht das zweite) Element des Arrays *feld* angesprochen wird! Aber auch das ist kein Problem: in C hat das erste Element eines Arrays den **Indexwert 0**. Assemblerprogrammierer werden erkennen, daß auch das nichts Besonderes ist. Wenn ich hier von 'Größe 10' spreche, heißt das nicht, daß der Array 10 Bytes belegt. Die in der Deklaration erscheinende Größenangabe gibt nur an, wieviele Einzelobjekte (Elemente) der Array umfaßt. Er hat in meinem Beispiel die Größe 10**sizeof(int)* Bytes.

Ich gebe jetzt einige Beispiele für die Deklaration und Initialisierung von eindimensionalen Arrays. Die Deklaration eines Arrays besteht immer aus einem Datentyp, einem Arraynamen und der Arraygröße zwischen eckigen Klammern. Eine Array-

initialisierung ist fakultativ; die Initialisierungswerte erscheinen nach einem Gleichheitszeichen und stehen meist (Initialisierung durch Strings bildet einen Sonderfall) zwischen geschweiften Klammern. Die Arraygröße muß ein Konstantenausdruck sein, also ein Ausdruck, der zur Compilezeit berechnet werden kann. Die Arraygröße darf nur bei formalen Argumenten und bei initialisierten Arrays fehlen. Beispiele für Deklarationen und Initialisierungen von Arrays:

```
/* Array von ints */
int iarray[3] =
{
   1, 2, 3
};
/* Array von floats -  ohne Groesse */
float farray[] =
{
  10.0, 12.2, -5.3E4
};
/* Array von char  */
char carray[3] = {'A', 'B', 'C'};

/* Strings:                    */
/* Array von char - Variante 1     */
/* Hier werden Inhalte eingetragen */
char carray[4] = "ABC";
/* Array von char - Variante 2 */
char carray[] = "ABC";

/* Array von Pointern auf char      */
/* Hier werden Adressen eingetragen */
char *parray[2] =
{
  "ABC",
  "DEF"
};
```

Der Compiler überprüft für initialisierte Arrays, ob die angegebene Arraygröße für die Elemente ausreicht. Wenn die Arraygröße bei initialisierten Arrays fehlt, wird sie aus den Initialisierungsangaben errechnet. Zur Laufzeit werden in C keine Indexwerte überprüft. Sie können also nach Herzenslust (falls dies sinnvoll ist und Ihr Betriebssystem mitmacht) die Arraygrenzen überschreiten. Bitte beachten Sie, daß sich Strings bei der Initialisierung anders verhalten als in normalen Befehlen. Normalerweise wird in einem Befehl wie *pchar="ABC"*; nicht der Stringinhalt, sondern die Ablageadresse des Strings zugewiesen. In Arrayinitialisierungen wie im Falle von *carray[]* (s. oben) wird tatsächlich der Stringinhalt zugewiesen. Im Falle des

Pointerarrays *parray[]* werden allerdings wieder Adressen zugewiesen; die Zuweisungslogik bei der Initialisierung wird also vom Datentyp gesteuert.

Denken Sie auch daran, daß in Arrays ein zusätzliches Byte für die binäre Null reserviert werden muß, wenn er Strings aufnehmen soll.

8.3 Die Parallelität von Arrays und Pointern

Warum beginnt die Zählung der Einzelvariablen (der Zellen, der Elemente) eines Arrays bei 0 und nicht bei 1? Jetzt ist es Zeit, die Parallelität von Arrays und Pointer vorzuführen. Dazu formuliere ich ein Programmteilstück von weiter oben etwas um:

```
int feld[10];      /* Deklaration Array */
int *afeld;        /* Deklaration Pointer */
  afeld = feld;
  if (*(afeld+2) == 17) *(afeld+2) = 18;
```

Ich habe einen Pointer *afeld* zusätzlich definiert, dem die Anfangsadresse des Arrays *feld* zugewiesen wird. Eine Zeile später wird wieder abgefragt, ob das dritte Element von *feld* den Wert 17 hat. Nehmen wir an, daß *feld* im Speicher an der Adresse 108 anfängt. Dann hat das dritte Element von feld die Adresse 112 (108, 110, 112 = 3 Elemente) und das ist genau *(108 + 2 * (sizeof(int)))*. Schon die Zeile *afeld = feld*; zeigt an, daß der Arrayname *feld* nichts anderes als ein Pointer ist – sonst könnte sein Wert nicht problemlos der Pointervariablen *afeld* zugewiesen werden. Statt *afeld = feld*; könnte ich übrigens auch *afeld = &feld[0]*; schreiben; der Arrayname *feld* ist eine Konstante mit dem Adreßwert des ersten Arrayelements.

Wenn Sie ein Element eines Arrays über einen Index in eckigen Klammern ansprechen, dann tut der C-Compiler genau das, was ich oben getan habe: er nimmt sich die Anfangsadresse des Arrays, addiert das Produkt aus Index (das ist der Wert des Ausdrucks zwischen den eckigen Klammern) und einem Ausdruck mit *sizeof* auf die Adresse und greift dann über die so errechnete Adresse auf die eigentlichen Daten zu.

Arraynamen können als Pointerkonstanten angesehen werden! Es ist wichtig, sich das zu merken. Sie sind keine üblichen Pointervariablen, denn einige der Operatoren, die sonst auf Pointer anwendbar sind, dürfen Sie auf Arraynamen nicht anwenden. So dürfen Arraynamen aus naheliegenden Gründen nicht auf der linken Seite eine Zuweisung vorkommen – damit würde da die Anfangsadresse des reservierten Speicherbereichs zerstört werden. Sie dürfen auf Arraynamen auch nicht den Inkrement- oder Dekrementoperator anwenden.

An den folgenden drei Beispielsprogrammen sollte die Parallelität von Arrays und Pointern endgültig klar werden. In allen drei Programmen wird die Länge eines übergebenen Strings – Strings werden in *char*-Arrays abgelegt und haben eine binäre Null

als Endekennung – berechnet. Für die Berechnung wird hier eine Prozedur verwendet, die in den drei Lösungen leicht unterschiedlich formuliert ist:

```
/*************************************

        Arrays und Pointer Loesung 1
            last update 10/07/87
       AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
        (C) 1987  by Markt & Technik

*************************************

Arrays und Pointer: Loesung 1
laenge() verwendet einen Array

*************************************/

#include <stdio.h>

char cfeld[80];

/* Die Laenge eines Strings berechnen */
int laenge(carr)
char carr[];
{
  int n;

  n = 0;
  while (carr[n++] != 0);
  return(n - 1);
}   /* end laenge */

main()
{
  int sl;

  printf("\nGib einen Satz ein:");
  gets(cfeld);
  sl = laenge(cfeld);
  printf("\nDer Satz ist %d Zeichen lang");
}   /* end main */
```

```
/****************************************

       Arrays und Pointer Loesung 2
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Arrays und Pointer: Loesung 2
laenge() verwendet einen Pointer

****************************************/

#include <stdio.h>

char cfeld[80];

/* Die Laenge eines Strings berechnen */
int laenge(cpoi)
char *cpoi;
{
  int n;

  n = 0;
  while (*cpoi++ != 0) n++;
  return(n);
}   /* end laenge */

main()
{
  int sl;

  printf("\nGib einen Satz ein:");
  gets(cfeld);
  sl = laenge(cfeld);
  printf("\nDer Satz ist %d Zeichen lang");
}   /* end main */
```

```
/****************************************

       Arrays und Pointer Loesung 3
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Arrays und Pointer: Loesung 3
laenge() verwendet einen Pointer
Adressierung dennoch a la array

****************************************/

#include <stdio.h>

char cfeld[80];

/* Die Laenge eines Strings berechnen */
int laenge(cpoi)
char *cpoi;
{
  int n;

  n = 0;
  while (cpoi[n++] != 0);
  return(n-1);
}   /* end laenge */

main()
{
  int sl;

  printf("\nGib einen Satz ein:");
  gets(cfeld);
  sl = laenge(cfeld);
  printf("\nDer Satz ist %d Zeichen lang");
}   /* end main */
```

Lösung 1 entspricht in etwa einer Schreibweise, die Sie in Pascal wählen würden, Lösung 2 dagegen ist C-spezifisch. Ungewöhnlich ist in Lösung 1 nur die Notierung *carr[]* ohne konkrete Indexangabe. Diese Schreibweise soll nur ausdrücken, daß *carr* im Unterprogramm als Array aufzufassen ist, dessen exakte Größe nicht bekannt ist.

In Lösung 2 wird das Unterprogramm ebenso wie in Lösung 1 mit dem Arraynahmen als Argument aufgerufen. Im Unterprogramm selbst jedoch wird ein Pointer verwendet, der dazu auch noch später inkrementiert – also im Wert verändert – wird. Der Sternoperator (in **cpoi++*) sowie der Inkrementoperator ++ wurden im Kapitel über die Operatoren bereits erläutert. Verständlich ist die Lösung nur, wenn man weiß, daß nicht der ganze Array übergeben wird, sondern nur die Anfangsadresse des Arrays (eben der Arrayname) und daß die Parameterübergabe üblicherweise über einen Stack erfolgt, also nach Kopie der Adresse von *cfeld* in einen speziellen Speicherbereich genannt Stack. Die Pointervariable *cpoi* in Lösung zwei liegt also nicht auf der gleichen Stelle wie *cfeld* und kann infolgedessen verändert werden, ohne daß *cfeld* dadurch etwas passiert!

Lösung 3 ist die mir am wenigsten sympathische Lösung. Dennoch ist diese Lösung in C zugelassen. Sie zeigt, daß man auch Pointervariablen (hier *cpoi*) syntaktisch wie Arrays einsetzen kann. Obwohl *cpoi* eine Variable ist, wird *cpoi* wie ein Arrayname – also eine Konstante – behandelt.

8.4 Mehrdimensionale Arrays

Ich habe bisher nur einstellige Arrays behandelt, also Arrays, die in Deklaration und Verwendung nur einen Index aufwiesen. In C sind aber auch mehrdimensionale Arrays erlaubt. Die Schreibweise und Verwendung entspricht genau den schon bekannten Konventionen. Statt einer eckigen Klammer müssen Sie halt mehrere eckige Klammernpaare verwenden. Beispiel für Deklaration und Verwendung:

```
/* ein mehrdimensionaler Array */
char schirm[25][81];
int i;
  for (i = 0; i < 25; i++)
    if (schirm[i][0] != 0)
         printf("\n%s",&schirm[i][0]);
```

Hier wird ein zweistelliger Array namens *schirm* definiert, der 25*80 Elemente hat. Mit einem solchen Array können Sie z.B. den Aufbau des Bildschirms definieren, an dem Sie gerade arbeiten. Er hat wahrscheinlich 25 Zeilen (erste Dimension) ä 80 Spalten (zweite Dimension). Die kleine *for*-Schleife könnte z.B. dazu dienen, den Schirminhalt zeilenweise auszugeben. Bitte beachten Sie, daß die Größenangabe für die zweite Dimension 81 (nicht 80) ist, weil – um im Beispiel zu bleiben – die binäre Null zusätzlich in der Bildschirmzeile untergebracht werden muß. Wenn ich diesen

Array *schirm* statt über *printf()* oder *puts()* über *putchar()* (also zeichenweise) ausgeben will, so benötige ich eine doppelte Schleife:

```
for (i = 0; n < 25; i++)
{
  for (j = 0; j < 80; j++)
  {
    if (schirm[i][j] == 0) break;
    putchar(schirm[i][j]);
  }
}
```

Wie Sie am Beispiel sehen, läuft die Schleife für die zweite Dimension (Variable *j*) schneller hoch.

Sie können sich einen zweidimensionalen Array (und analog dazu beliebige mehrdimensionale Arrays) vorstellen als einen Array von Adressen von eindimensionalen Arrays. In C müssen nicht immer alle Dimensionen bei der Verwendung eines Arrays angegeben werden. Im Beispielsprogramm *bubble.c* weiter unten sehen Sie, daß ich einmal einen zweidimensionalen Array mit nur einem eckigen Klammernpaar anspreche. Wie sieht nun die Adreßrechnung aus, die der C-Compiler anstellt? Im erzeugten Code wird ja eine Adresse benutzt – Arrays gibt es dort nicht. Die Adreßrechnung des Compilers sieht ungefähr so aus:

```
/* Ausgangsausdruck */
schirm[i][j]
/* aequivalenter Ausdruck mit */
/* expliziter Adressrechnung  */
*(schirm + (i * 81 * sizeof(char)) + j)
```

Wenn Sie mit einem Binärdebugger in ein Programm schauen, werden Sie solche Adreßrechnungen wiedererkennen.

Bei der Initialisierung eines mehrdimensionalen Arrays müssen die Initialisierungsausdrücke ähnlich eingebettet werden wie die Schleifen im obigen Beispiel:

```
/* Initialisierung eines zweidimensionalen Arrays */
int iarray[2][2] =
{
  { 3, 4},            /* [0][0] und [0][1] */
  { 5, 6}             /* [1][0] und [1][1] */
};
```

Bei der Initialisierung komplexerer C-Objekte wie diesem ist es günstig, die Struktur der Objekte durch Einrückungen hervorzuheben.

Das nachfolgende Beispielsprogramm zeigt einen bekannten aber ineffizienten Sortieralgorithmus: *bubblesort*. Ich verwende hier zweidimensionale *char*-Arrays, weil ich Wörter einlese und sortiere. Wenn Sie Zahlen sortieren wollen, reicht ein eindimensionaler Array. Sie sollten sich das Beispiel genauer ansehen, weil hier auch Operatoren wie *sizeof* und der Adreßoperator auf einen zweidimensionalen Array angewandt werden. Außerdem erscheint der Array im Aufruf von Prozeduren der C-Bibliothek.

```
/*************************************

 Demonstration fuer zweistellige Arrays
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

*************************************

Demonstrationsprogramm fuer zweidimensionale Arrays

*************************************/

#include <stdio.h>

char wort[10][20],hfeld[20];

main()
{
  int diff,dim1,dim2,maxwort;
  int i,n,m,klflag;

  maxwort = 0;
  dim2    = sizeof(wort[0]);
  dim1    = sizeof(wort) / dim2;

  /* Woerter einlesen und in wort ablegen  */
  /* Eingabe einer leeren Zeile = Ende    */
  while (maxwort < dim1)
    {
     printf("\nnaechstes Wort:");
     gets(&wort[maxwort][0]);
     if (strlen(&wort[maxwort][0]) == 0) break;
     maxwort++;
    }   /* while maxwort ... */

  /* Jetzt aufsteigend sortieren */
  for (n = 0; n < maxwort; n++)
    {
     for (m = n+1; m < maxwort; m++)
       {
        klflag = 0;
        /* Stringvergleich (auch strcmp moeglich) */
        for (i = 0; i < dim2; i++)
          {
           diff = wort[m][i] - wort[n][i];
           if (diff < 0)
```

```
            {
             klflag = 1;
             break;
            }
          if (diff > 0)          break;
          if (wort[n][i] == 0) break;
        }   /* for i = ... */
      if (klflag)
        {
         /* Wort an Pos. m ist kleiner als Wort an Pos. n */
         /* Positionen n und m in Array vertauschen       */
         /* Pos. n in hfeld retten
         strcpy(hfeld,&wort[n][0]);
         strcpy(&wort[n][0],&wort[m][0]);
         strcpy(&wort[m][0],hfeld);
        }
      }   /* for m .... */
    }  /* for n.... */

  /* Jetzt sortiertes Ergebnis ausgeben */
  for (n=0; n < maxwort; n++)
    printf("\n%s",&(wort[n][0]));
}   /* end main */
```

9 Strukturen und Verbunde

9.1 Begriffbestimmungen

Unter einer Struktur ('record' in anderen Programmiersprachen) versteht man eine Folge von ein oder mehreren Elementen, die inhaltlich zusammengehören und über den Strukturnamen und den Elementnamen ansprechbar sind. Das Beispiel einer Personalakte (stark verkürzt) wird oft zur Erläuterung des Konzepts Struktur herangezogen. In C könnte eine Personalakte z.B. so deklariert werden:

```
struct
{
  char fname[20];
  char vname[20];
  char strasse[20];
  char wohnort[20];
  long verdienst;
} personalakte;
```

In der Deklaration einer Struktur muß das Schlüsselwort *struct* erscheinen. Wie Sie sehen, kann eine Struktur durchaus aus Variablen und anderen Objekten mit unterschiedlichem Typ bestehen. In unserem Beispiel besteht die Struktur *personalakte* aus fünf Elementen (so heißen die Mitglieder einer Struktur): aus dem Familiennamen *fname*, aus dem Vornamen *vname*, aus dem Straßennamen *strasse*, aus dem Wohnort *wohnort* und aus der Höhe des Gehalts *verdienst*. In meinem Beispiel ist nur *verdienst* eine Variable im eigentlichen Sinn. Arrays (die anderen vier Elemente) sind Folgen vor Variablen. Sie sehen schon an diesem einfachen Beispiel, daß Strukturen auch Elemente mit komplexerem Charakter beinhalten können; auch Strukturen, Verbunde und Arrays als Elemente von Strukturen sind erlaubt.

Der zusammengesetzte Datentyp **Verbund** ist sehr stark mit dem Datentyp Struktur verwandt. Während jedoch die Elemente einer Struktur nacheinander angeordnet sind, beginnen alle Elemente eines Verbunds auf der gleichen Speicheradresse. Verbunde dienen dazu, das gleiche Datenobjekt unter unterschiedlichen Elementnamen anzusprechen. Das nachstehende Beispiel variiert die obige Deklaration der Personalakte. Der Array *wohnort* wird durch den Verbund *ortsangabe* ersetzt.

```
/* Postleitzahl und Wohnort im Verbund */
struct
{
  char fname[20];
  char vname[20];
  char strasse[20];
  union
  {
    char plz[4];
    char wohnort[25];
  } ortsangabe;
  long verdienst;
} personalakte;
```

```
/* Zugriff */
strcpy(personalakte.ortsangabe.plz,"6642");
printf("\n%s",personalakte.ortsangabe.plz);
strcpy(personalakte.ortsangabe.wohnort,
        "6641 Harlingen");
printf("\n%s",personalakte.ortsangabe.wohnort);
```

In der Deklaration eines Verbunds erscheint das C-Schlüsselwort *union*. Wie Sie sehen werden, entspricht der Zugriff (Punktoperator und Pfeiloperator!) vollständig dem Zugriff auf Strukturen. Das Beispiel soll zeigen, wie Sie auf die vier ersten Positionen eines Strings ohne Einzelzeichenverarbeitung zugreifen können. Der Inhalt von *plz* ist vollständig im Inhalt von *wohnort* enthalten: er belegt die ersten 4 Bytes von *wohnort*. Mit dem ersten Aufruf von *strcpy()* werden vier Bytes in *plz* eingetragen. Das durch *strcpy()* einkopierte Nullbyte belegt die Arrayposition 4 von *wohnort*, geht also über *plz* hinaus. Statt nur *plz* im ersten *printf()* auszugeben könnte ich auch *wohnort* ausgeben – das Ergebnis wäre gleich.

Wenn Sie den *sizeof*-Operator auf den Verbund *ortsangabe* ansetzen, werden Sie den Wert 25 erhalten – also die Länge des größten Elements des Verbunds. Die Größe eines Verbundes entspricht immer der Größe des längsten Elements. Das Verbundkonzept entspricht dem *EQUIVALENCE*-Konzept In Fortran bzw. *DEFINED* in PL/1.

Im obigen Beispiel waren die Elemente des Verbunds vom gleichen Datentyp. Sehr häufig werden Sie jedoch Verbunde sehen, deren Elemente unterschiedliche Datentypen aufweisen. Das nachfolgende Beispiel zeigt einen Verbund, der dazu benutzt wird, die Einzelbytes einer *double*-Variablen auszugeben:

```
/* Ausgabe der Einzelbytes einer double-Zahl */
union
{
   char dc[8];
   double dv;
} dverbund;
int n;

dverbund.dv = -18.75;
for (n = 0; n < 8 ; n++)
  printf("%02x",dverbund.dc[n]);
```

Für Feinde von Verbunden will ich eine inhaltlich gleiche, syntaktische aber total verschiedene Lösung des gleichen Problems angeben. Ich benutze hier eine *char*-Variable *hadr*, der ich die Anfangsadresse der *double*-Zahl zuweise:

```
/* Umgehung des Verbundes              */
/* Ausgabe der Bytes einer double-Zahl */
unsigned char *hadr;
double dv;
int n;
   hadr = &dv;
   for (n=0; n < sizeof(double); n++)
     printf("%02x",*hadr++);
```

9.2 Deklaration und Initialisierung

Bei der Deklaration von Strukturen und Verbunden gibt es mehrere Varianten. Wichtig ist zunächst, daß das Schlüsselwort *struct* oder *union* verwendet wird. Dann muß unterschieden werden zwischen dem **Strukturmuster** (structure tag) und der eigentlichen Struktur. Das Strukturmuster gibt nur den Aufbau der Struktur oder des Verbundes an; es belegt selbst keinen Platz. Der Name des Strukturmusters kann als Abkürzung verwendet werden, wenn das Muster einmal deklariert wurde. Der Name des Strukturmusters erscheint unmittelbar hinter den Schlüsselwörtern *union* oder *struct*. Der Strukturname erscheint nach der Deklaration der Einzelelemente.

Mir persönlich ist die Deklarationsvariante mit dem *typedef*-Operator am liebsten, da sie am übersichtlichsten ist und die Schlüsselwörter *struct* und *union* nach der Deklaration des benutzereigenen Datentyps überflüssig macht. Die folgenden Beispiele zeigen mehrere Varianten der Deklaration der gleichen Struktur:

```
/* Variante 1: ohne Strukturmuster */
struct
{
   char name[20];
   long verdienst;
} akte;

/* Variante 2: Strukturmuster und Struktur */
/* werden separat deklariert               */
struct AKTE          /* Strukturmuster */
{
  char name[20];
  long verdienst;
};
struct AKTE akte;     /* Dekl. Struktur */

/* Variante 3: Strukturmuster und Struktur werden */
/* gleichzeitig deklariert                        */
struct AKTE           /* Strukturmuster */
{
  char name[20];
  long verdienst;
} akte;               /* Struktur */

/* Variante 4: Deklaration Strukturmuster über typedef */
typedef struct AKTE
{
  char name[20];
  long verdienst;
} MEINE_AKTE;              /* Name des benutzereigenen Typs
MEINE_AKTE akte;           /* Deklaration Struktur */
```

Bitte schauen Sie sich diese – vielleicht verwirrenden – Beispiele genau an. In Variante 2 wird durch die erste Deklaration kein Speicherplatz reserviert, da nur das Strukturmuster deklariert wird. In Variante 4 wird durch die Deklaration des benutzereigenen Datentyps über *typedef* ebenfalls kein Platz reserviert.

Bei der Initialisierung einer Struktur erscheinen die initialisierenden Werte zwischen geschweiften Klammern. Eingebettete Unterstrukturen oder Verbunde sollten durch geschweifte Klammern zusammengefaßt werden; bei Strukturarrays sollten die Einzelstrukturen ebenfalls geklammert werden. Das folgende Beispiel zeigt die Deklaration und Initialisierung eines Arrays von Strukturen der Dimension 2:

```
struct
{
  char name[20];
  long verdienst;
} akte[2] =
{
  { "maier",20000L},
  { "mueller",60000L}
};
```

9.3 Zulässige Operationen

Auf die Elemente einer Struktur oder eines Verbunds sind alle Operationen erlaubt, die sonst für 'normale' Datenobjekte gelten. Ich kann diesen Elementen also einen Wert zuweisen, sie initialisieren, sie vergleichen, sie verändern, sie als Argumente an eine Prozedur übergeben, wenn der Datentyp dies erlaubt. Die Elemente einer Struktur oder eines Verbunds werden allerdings etwas anders als normale Variablen angesprochen: zur Auswahl eines Strukturelements wird der Punktoperator benutzt. Hier drei Beispiele für den Zugriff auf Elemente der Struktur aus Kap. 9.1:

```
strcpy(personalakte.fname,"Mueller");
personalakte.vname[0]  = 0;
personalakte.verdienst = 50000L;
```

Wenn statt des Strukturnamens ein Pointer auf eine Struktur als Haupteinstieg benutzt wird, müssen Sie den Pfeiloperator statt des Punktoperators verwenden. Das gleiche gilt für Verbunde. Das folgende Beispiel zeigt den Gebrauch des Pfeiloperators und die Deklaration eines Pointers auf eine Struktur:

```
/* Pfeiloperator statt Punktoperator */
struct
{
   char fname[20];
   char vname[20];
} name1, /* Deklaration Struktur */
 *pname; /* Pointer auf Struktur */

/* Zugriff */
pname = &name1;
strcpy(pname->fname,"Huckert");
strcpy(pname->vname,"Edgar");
```

Strukturen werden von den C-Compilern wie zusammenhängende Speicherbereiche betrachtet. Die Elemente einer Struktur erscheinen im angelegten Speicher in der deklarierten Reihenfolge. In meinem ersten Beispiel wird also das Element *verdienst* physikalisch nach dem Element *wohnort* erscheinen. Es ist ebenso üblich, daß für Strukturen bisweilen mehr Platz reserviert wird, als nötig wäre. Die Strukturelemente werden – so sagt der Fachmann – auf Ganzwortgrenzen ausgerichtet. Sie sollten daran denken, wenn Sie einmal mit einem Debugger in eine Struktur schauen und dabei binären Nullen begegnen, die Ihnen seltsam vorkommen.

Auf Strukturen und Verbunde als Ganzes können Sie nur wenige Operationen anwenden: Sie können nur die Adresse einer Struktur ermitteln mittels des Adreßoperators und die Elemente einer Struktur mit dem Punktoperator ansprechen. Bitte beachten Sie, daß nach K&R Strukturen nicht direkt einander zugewiesen werden können. Sie dürfen sie auch nicht direkt miteinander vergleichen oder als Argumente an eine Prozedur übergeben. Funktionen können auch keine Strukturen als Werte liefern. In K&R wird schon darauf hingewiesen, daß sich das vielleicht einmal ändern könnte. Tatsächlich wird die ANSI-Norm für C (s. Kap. 16) einige dieser Restriktionen aufheben. Ich behandle hier diese Erweiterungen nicht, da es noch zu viele C-Compiler gibt, die dem K&R Standard folgen. Aber keine Bange: Sie können mit diesen Restriktionen – Sie werden es gleich sehen – gut leben.

Funktionen, die Strukturen als Werte liefern, werden im Kap. 10 behandelt. Zuweisungen und Vergleiche von Strukturen und Verbunden sind sehr leicht mit C-Standardmitteln zu bewerkstelligen. Das folgende Beispiel demonstriert die Zuweisung von kompletten Strukturen über eine Prozedur. Der Vergleich von Strukturen (nur der Vergleich auf Identität ist wirklich sinnvoll) kann ebenso einfach programmiert werden:

```
typedef struct
{
  int xyz;
  int uvw;
} DEMO;
DEMO struct1,struct2;

/* Kopiert nbytes Bytes von quelle nach ziel */
structcpy(ziel,quelle,nbytes)
unsigned char *ziel,*quelle;
int nbytes;
{
  while (nbytes-- > 0)
    *ziel++ = *quelle++;
}   /* end structcpy */
```

```
main()
{
  /* .......... */

  /* Zuweisung struct1 := struct2 */
  structcpy(&struct1,&struct2,sizeof(DEMO));
}   /* end main */
```

Der Adreßoperator macht es zusammen mit dem *sizeof*-Operator wieder einmal möglich. Wer jetzt auf die Idee kommt, statt meines Unterprogramms *structcpy()* die Bibliotheksroutine *strcpy()* zu verwenden, hat verloren! *strcpy()* stoppt nämlich beim ersten Auftreten einer binären Null – und die kann sowohl als Inhalt eines Strukturelements wie auch als Füller zwischen Strukturelementen auftreten! Statt *strcpy()* gibt es in einigen C-Bibliotheken auch Funktionen wie *memcpy()* oder ähnlich, die keine Rücksicht auf die binäre Null nehmen. Wie Sie sehen ist es jedoch keine große Tat, sich solche Funktionen selbst (und dazu portabel) zu schreiben.

9.4 Selbstbezügliche Strukturen

Der Ausdruck 'selbstbezügliche Struktur' (self referential structure) ist eine etwas unglückliche Bezeichnung, die sich in der C-Welt allerdings durchgesetzt hat. Gemeint sind Strukturen, in denen Verweise (Pointer) auf Strukturen des eigenen Typs – also mit dem gleichen Strukturmuster – vorkommen. Solche selbstbezüglichen Strukturen werden vor allem dazu verwendet, **verkettete Listen, Bäume** etc. zu definieren. Ein typisches Beispiel für die Anwendung verketteter Listen sind dynamische Arrays: das folgende Beispielsprogramm demonstriert dies. Beachten Sie bitte in der *typedef*-Deklaration für *KETTE* die Zeile *struct kette *next*;. Sie enthält den Bezug auf das eigene Strukturmuster. Der Ausdruck 'dynamischer Array' ist im Sinne von C nicht ganz korrekt, weil die Elemente nicht über den []-Operator angesprochen werden, sondern über Pointer. Der so gebildete Array heißt 'dynamisch', weil er beliebig lang ist: ich kann jederzeit eine weitere Struktur an das letzte Glied der Kette hängen. Der Array ist in meinem Beispiel gleich doppelt dynamisch; er hat beliebig viele 'Elemente', und die Elemente sind beliebig lang. Jede Struktur und jeder String wird nämlich über *malloc()* (s. Kap. 12) angelegt. Hier wird nur Vorwärtskettung angewendet, d.h. eine Struktur hat nur einen Pointer auf ihren Nachfolger, da ich sequentiell nur in eine Richtung suchen will. Solche dynamischen Arrays werden vor allem dazu benutzt, um Daten einzulesen, deren Anzahl nicht bekannt ist. Das Programm liest beliebige Zeilen ein, legt sie als verkettete Liste ab und sucht dann darin.

```
/****************************************

            Verkettete Listen
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Demonstrationsprogramm fuer verkettete Listen

****************************************/

#include <stdio.h>

typedef struct kette
{
  char *pzeile;
  struct kette *next;
} KETTE;
KETTE *start,*zuletzt;

extern char *malloc(),*free();

/* legt einen Eintrag in der verketteten Liste an */
/* setzt start und zuletzt                        */
mache_eintrag(str)
char *str;
{
  KETTE *pstruct;
  char *cadr;
  /* exakt grossen Array fuer str anlegen */
  cadr = malloc(strlen(str)+1);
  strcpy(cadr,str);
  /* jetzt Struktur allokieren */
  pstruct = malloc(sizeof(KETTE));
  /* Pointer auf Inhalt */
  pstruct->pzeile = cadr;
  /* Pointer auf Nachfolger */
  pstruct->next = NULL;
  if (start != NULL)
    zuletzt->next = pstruct;
  else start = pstruct;
  zuletzt = pstruct;
}   /* end mache_eintrag */

main()
{
  char zeile[100];
  KETTE *hadr;

  /* Anlegen - Ende = CR */
  start = zuletzt = NULL;
  while (1)
```

```
    {
     printf("\nnaechste Zeile:");
     gets(zeile);
     if (strlen(zeile) == 0) break;

     mache_eintrag(zeile);
    }   /* while 1 */

  /* Suchen - Ende = CR */
  while (1)
    {
     printf("\nSuchstring:");
     gets(zeile);
     if (strlen(zeile) == 0) break;
     hadr = start;
     while (hadr != NULL)
       {
        if (strcmp(hadr->pzeile,zeile) == 0) break;
        hadr = hadr->next;
       }   /* while hadr != NULL */
     if (hadr != NULL) printf("\t\t\t --- gefunden");
     else              printf("\t\t\t --- nicht gef.");
    }   /* while 1 */

  /* allokierten Platz wieder freigeben */
  /* nicht unbedingt noetig              */
  hadr = start;
  while (hadr != NULL)
    {
     zuletzt = hadr->next;
     free(hadr->pzeile);
     free(hadr);
     hadr = zuletzt;
    }
}   /* end main */
```

Die folgende Zeichnung soll das Prinzip der verketteten Listen veranschaulichen. Ich nehme dabei an, daß drei Namen, nämlich 'Gisela', 'Philipp' und 'Martin' eingelesen und in die Kette eingereiht wurden. Die Pfeile sollen Pointer darstellen. *start* ist der Pointer auf das erste Kettenelement. Das letzte Element der Kette weist den leeren Nachfolgepointer auf, nämlich NULL.

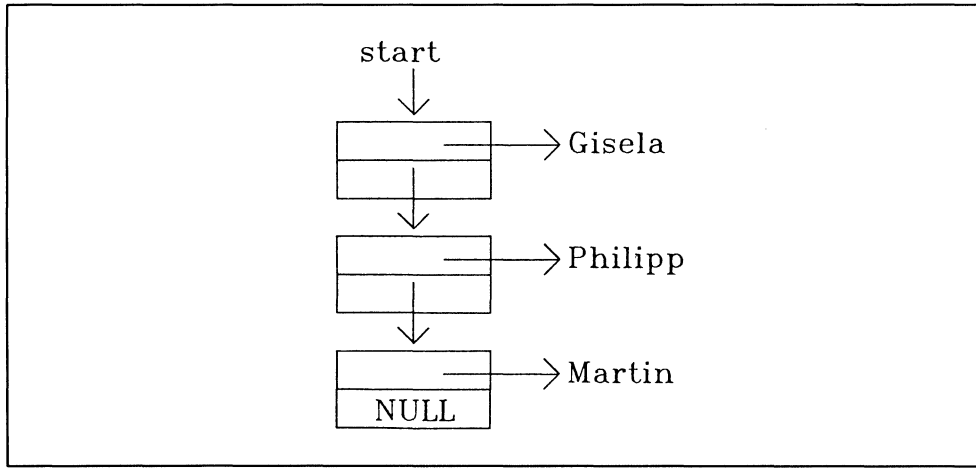


Bild 9.1: *Verkettete Liste*

9.5 Ein Testprogramm für Strukturen

Das folgende Testprogramm zeigt den Umgang mit Strukturen. Es verwendet das Beispiel einer Personalakte – oder besser gesagt: eines Arrays von Personalakten. Aus einer stringähnlichen Eingabezeile (keine feste record-Struktur wie in der kommerziellen DV üblich) werden Felder extrahiert und in der Prozedur *readstruct()* in die Elemente der C-Struktur eingefüllt. Eine Schleife in *main()* wertet dann die gelesenen Daten aus. Im Gegensatz zum vorigen Beispiel haben hier fast alle Strukturelemente zusammengesetzte Datentypen, nämlich Arrays. Zudem wird hier ein Array von Strukturen statt einer verketteten Liste verwendet. Der Array muß exakt dimensioniert werden, während die verkettete Liste aus (nahezu) beliebig vielen Kettenelementen bestehen konnte, die dynamisch über *malloc()* angelegt wurden. Das Nebeneinander von Array und Struktur führt zu einer kompliziert aussehenden, letztlich aber leicht verständlichen Kombination von Arrayelementoperator [] und Punktoperator. Sie sollten sich also die Schreibweise für den Zugriff auf die Einzelelemente der Struktur PERSONAL genau ansehen:

Eingabedaten (Beispiel):

```
huckert,edgar,hauptstrasse 17,6641 harlingen,40000
maier,richard,am brothaus 22,6640 merzig,50000
mueller,adalbert,waldweg 21,6638 beckingen,55000
```

Programmabdruck:

```
/****************************************

 Demonstrationsprogramm fuer Strukturen
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Einlesen und Auswerten von Personalakten

****************************************/

#include <stdio.h>

#define LF       0x0a
#define LSTR     20
#define MAXPERS  100

/* Aufbau einer Personalakte */
typedef struct
{
  char fname[LSTR];    /* Familienname */
  char vname[LSTR];    /* Vorname */
  char strasse[LSTR];
  char wohnort[LSTR];
  long verdienst;    /* Jahresverdienst */
} PERSONAL;
PERSONAL pakte[MAXPERS];

/* Legt einen durch Komma oder LF begrenzten String */
/* in ziel ab                                       */
/* Liefert die Startadresse des Naechsten            */
char *getfield(quelle,ziel)
char *quelle,*ziel;
{
  while (*quelle != ',')
    {
     if (*quelle == LF) break;
     *ziel++ = *quelle++;
    }
  *ziel = 0;
  return(quelle+1);
}   /* end getfield */

/* Liest den naechsten Datensatz fuer die   */
/* Personalakte ein                         */
/* Liefert die Adresse des abgelegten       */
/* Datensatzes oder NULL                    */
/* Alle Eintraege im Datensatz muessen      */
/* vollstaendig sein                        */
/* Jeder Eintrag ist vom folgenden durch    */
/* ein Komma getrennt                       */
```

```
PERSONAL *readstruct(ifil,npakte)
FILE *ifil;
/* Nummer der naechsten freien Personalakte */
int npakte;
{
  char zeile[80],*zadr;

  if (fgets(zeile,80,ifil) == NULL) return(NULL);
  zadr = getfield(zeile,pakte[npakte].fname);
  zadr = getfield(zadr,pakte[npakte].vname);
  zadr = getfield(zadr,pakte[npakte].strasse);
  zadr = getfield(zadr,pakte[npakte].wohnort);
  sscanf(zadr,"%ld",&pakte[npakte].verdienst);
  return(&pakte[npakte]);
}   /* end readstruct */

main()
{
  int n,maxpers;
  long lohnsum;
  char filna[100];
  FILE *ifil;

  maxpers = 0;  /* zaehlt die Personalakten */

  /* Einlesen der Daten */
  printf("\nDatenfile:");
  gets(filna);
  if ((ifil = fopen(filna,"r")) == NULL) exit(1);
  while (1)
    {
     if (readstruct(ifil,maxpers) == NULL) break;
     if (maxpers++ >= MAXPERS) break;
    }

  /* Personalakten verarbeiten */
  lohnsum = 0;
  for (n = 0; n < maxpers; n++)
    {
     printf("\nAngestellter: %15s  Lohn:%ld", pakte[n].fname,
                                              pakte[n].verdienst);
     lohnsum += pakte[n].verdienst;
    }
  printf("\n\nLohnsumme: %ld",lohnsum);
  fclose(ifil);
}   /* end main */
```

10 Prozeduren

10.1 Grundlegendes zu Prozeduren

Anders als in anderen Programmiersprachen gibt es in C keinen Unterschied zwischen **Prozeduren** und **Funktionen**. Jede Prozedur in C liefert einen Wert, ist also eine Funktion. Ich werde also die beiden Begriffe 'Prozedur' und 'Funktion' gleichbedeutend benutzen. Ab und zu fällt auch der Begriff 'Routine'; auch er wird als Synonym für 'Prozedur' verwendet. Ich könnte auch den Begriff 'Unterprogramm' verwenden: in C sind alle Prozeduren außer *main()* Unterprogramme; *main()* ist – wie der Name sagt – das Hauptprogramm. Programme in C bestehen aus einer Sammlung von Prozeduren.

Wozu sind Prozeduren gut? Sie dienen einerseits dazu, immer wiederkehrende, verwandte Aufgaben zu einem Programmgebilde zusammenzufassen. Andererseits werden sie oft auch nur verwendet, um ein Programm in logisch identifizierbare Blöcke zu unterteilen. Beide Aufgaben sind eng miteinander verwandt.

Wenn man bestimmte, immer wieder benötigte Codestücke zu einer Prozedur zusammenfaßt, wird insgesamt Programmplatz gespart. Als Nebeneffekt – das wurde schon angesprochen – werden die Gesamtprogramme übersichtlicher. Übrigens haben Prozeduren nicht nur Vorteile: die Laufzeit von Programmen kann wachsen. Dieser Nachteil kann jedoch in den meisten Programmen vernachlässigt werden. Bitte beachten Sie den Unterschied zwischen Prozeduren und Makros (s. Kapitel über den Präprozessor): Prozeduren erscheinen als Codestück nur einmal in einem Programm, Makros werden immer durch den Präprozessor expandiert – die gleiche Codestrecke erscheint also u.U. sehr oft in einem Programm, das dem Compiler vorgesetzt wird. Prozeduren werden zur Laufzeit über den Funktionsnamen aufgerufen; ein Makroname (die 'Nennung', nicht der 'Aufruf' eines Makros) ist schon zur Compilezeit nicht mehr im Programm vorhanden.

Ich habe oben bewußt von 'verwandten' Aufgaben gesprochen. Prozeduren können nämlich durch die Übergabe von Argumenten parametrisiert werden, also leicht an die jeweils konkrete Aufgabe angepaßt werden. Statt **Argument** ist auch der Terminus **Parameter** gebräuchlich. Man unterscheidet zwischen **formalen** Argumenten – das sind die, die in der Definition einer Prozedur erscheinen – und **aktuellen** Argumenten – das sind die, die tatsächlich im Aufruf verwendet werden. Ein Beispiel:

```
/*************************************

          Quadratzahlberechnung
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

*************************************

Errechnet eine Quadratzahl

*************************************/

#include<stdio.h>

int quadrat(zahl)
int zahl;
{
   return(zahl * zahl);
}   /* end quadrat */

main()
{
  printf("\nQuadratzahl von 10 = %d",quadrat(10));
}   /* end main */
```

Im Aufruf der Prozedur stellt 10 das aktuelle Argument dar, *zahl* ist das formale Argument. Die Verwendung von parametrisierten Prozeduren ermöglicht den Aufbau von Modulbibliotheken. Solche 'Module' sind inhaltlich selbständige Prozeduren, die, wenn sie einmal geschrieben sind, wie ein 'Werkzeugkasten ' (toolbox) verwendet werden können. Ihr interner Aufbau interessiert nicht mehr; sie gelten nach außen hin als abgeschlossen. Modulbibliotheken können als Quellbibliotheken oder als Objektbibliotheken organisiert werden. Im Falle von Quellbibliotheken wird der angesprochene Code über #include-Befehle des Präprozessors vor dem eigentlichen Compilerlauf einkopiert. Im Falle von Objektbibliotheken übernimmt der Linker die Aufgabe, die benötigten Module (dann in vorkompilierter Form) dazuzubinden.

In Fortran und PL/1 erkennt man Unterprogramme daran, daß sie über *CALL* aufgerufen werden. Funktionen in diesen beiden Sprachen und in C werden nur durch die Namensnennung (z.B. *x=sqrt(2.0)*) aufgerufen, meist auf der rechten Seite von Zuweisungen oder als Argumente in anderen Unterprogrammen oder Funktionen. In C gibt es kein CALL-Schlüsselwort, so daß die Analogie zwischen Funktion und Unterprogramm syntaktisch untermauert ist. Funktionen und Unterprogramme

(Prozeduren allgemein) werden in C einfach durch Namensnennung gefolgt von (eventuell null) Argumenten in runden Klammern aufgerufen. Die runden Klammern dürfen – auch in parameterlosen Prozeduren – nicht weggelassen werden! Erst durch die runden Klammern nach einem Prozedurnamen wird ein Prozeduraufruf identifiziert. Die runden Klammern stellen den **Funktionsaufrufoperator** dar. Sie machen aus einem Pointer (Konstante oder Variable) einen Funktionsausdruck. Funktionen sind Prozeduren, die einen Wert liefern. In der Tat liefert in C jede Prozedur einen Wert – selbst *main()!* Auch wenn Sie nicht explizit einen Rückgabewert beim Verlassen einer Prozedur angeben – der C-Compiler generiert einen Rückgabewert, ob Sie es nun wollen oder nicht. Es steht Ihnen allerdings frei, ob Sie diesen Wert erfahren wollen oder nicht. Sie müssen also Funktionen nicht unbedingt syntaktisch als Funktionen kennzeichnen, indem Sie sie wie eine Variable verwenden. Nichts in C hindert Sie daran, eine Funktion wie die Quadratwurzel *sqrt()* nicht als Funktion zu verwenden – ob dies sehr sinnvoll ist, ist eine andere Frage.

Der Wert einer Funktion wird über den *return*-Befehl der rufenden Prozedur mitgeteilt. Die meisten Programmierer – ich ebenfalls – schreiben den Rückgabewert hinter *return* in runden Klammern. Das ist nicht nötig, hat sich aber so eingebürgert. Der Rückgabewert kann eine Konstante, eine Variable, ein komplexer Ausdruck oder ein Funktionsaufruf sein; Sie dürfen nur Ausdrücke als Werte verwenden, die einen einfachen Typ haben, also keine Arrays und keine Strukturen, die ja zusammengesetzte Typen darstellen. Weiter unten zeige ich, wie Sie mit dieser Einschränkung leben können. Typische *return*-Befehle sind also:

```
return;                    /* kein Rueckgabewert */
return(17);                /* Konstante */
return(var1);              /* Wert der Variablen */
return(&var1);             /* Adresse */
return(var1 + var2);       /* Wert des Ausdrucks */
return(xfunction(17,32));  /* Wert von xfunction */
```

return-Befehle dürfen überall in einer Prozedur bzw. einer Funktion vorkommen, nicht erst am Ende der Prozedur. Wenn Sie *return* ohne Wert verwenden, wird trotzdem ein Wert an die rufende Prozedur übergeben – er ist allerdings undefiniert. Deshalb sollte ein *return* ohne Wertangabe nie in Funktionen erscheinen. Der C-Compiler wird Ihnen allerdings keinen Fehler melden, wenn Sie in Funktionen trotzdem *return* ohne Rückgabewert verwenden. Natürlich sollte der Typ des Ausdrucks hinter *return* mit dem deklarierten Typ der Prozedur übereinstimmen.

Sie sollten nicht den Fehler machen, den Funktionsnamen wie eine Variable zu behandeln und an ihn einen Wert zuzuweisen, wie Sie es vielleicht von Funktionen in Pascal oder Fortran gewöhnt sind. Funktionsnamen sind **Adreßkonstanten** in C – sie dürfen also nicht auf der linken Seite einer Zuweisung vorkommen. Sie sind Rwerte, nicht Lwerte. Funktionsnamen sind nichts anderes als die Adresse im Speicher, an der der Funktionskörper beginnt. Das erklärt auch, weshalb der Adreßoperator nicht

auf Funktionsnamen angewandt werden sollte (die meisten Compiler erlauben das dennoch): der Adreßoperator ist überflüssig, weil ja schon eine Adresse vorliegt!

Die Definition einer Prozedur besteht aus mehreren Teilen, deren Reihenfolge eingehalten werden sollte:

- Deklaration der Prozedur: umfaßt den Typ der Funktion, den Namen der Prozedur und die Liste der formalen Argumente.
- Deklaration der Argumente
- öffnende geschweifte Klammer
- Deklaration der lokalen Variablen
- Prozedurkörper: eine Folge von C-Befehlen, die das definieren, was die Prozedur tun soll.
- schließende geschweifte Klammer

Sie sollten die Syntax der Prozedurdeklaration – also der ersten Zeile einer Prozedur – beachten: nach der Angabe des Typs, des Namens und der formalen Argumente erscheint normalerweise kein Semikolon. Wenn Sie trotzdem hinter die runde Klammer ein Semikolon setzen, dann haben Sie die leere Prozedur definiert –Ihre Prozedur tut nämlich nichts! Auf die runden Klammern können Sie nicht verzichten, auch wenn keine Argumente vorhanden sind. Der Prozedurkörper wird von geschweiften Klammern eingeschlossen. Eine typische C-Prozedur, die alle genannten Elemente enthält, ist die nachfolgende Beispielsprozedur:

```
int beispiel(x,y,z)     /* Deklaration Prozedur  */
int x,y,z;              /* Deklaration Argumente */
{
int summe;              /* Dekl. lokale Variable */
  summe = x + y  + z;   /* Beginn Prozedurkoerper */
  return(summe);
}   /* end beispiel */
```

Anders als in Pascal wird der Aufruf von Prozeduren nicht anhand der Definition der Prozedur überprüft: niemand hindert Sie also daran, eine Prozedur, die mit drei formalen Argumenten definiert ist, mit vier oder nur zwei aktuellen Argumenten aufzurufen. Sie werden nicht daran gehindert, falsche Argumenttypen zu verwenden. Statt *x=sqrt(2.0)* können Sie ungestraft *x=sqrt(2)* schreiben. Im harmlosen Fall liefert dieser Aufruf ein unsinniges Resultat. Vielleicht stürzt aber auch das Programm ab... Sie werden auch nicht daran gehindert, den Typ des Rückgabewerts von Funktionen zu verbiegen, etwa eine Funktion vom Typ *char* * als Funktion vom Typ *int* aufzurufen. Die Freiheit, die C Ihnen hiermit einräumt, kann katastrophale Fehler zur Folge haben. Aber C ist für Leute gedacht, die wissen was sie tun. Und vor allem: es gibt sinnvolle Anwendungen für das Ausnutzen dieser Freiheit. Das bekannteste Beispiel ist die Prozedur *printf():* sie erlaubt beliebig viele Argumente. Es wäre also sinnlos,

eine Überprüfung der Zahl der Argumente zur Compilezeit vorzunehmen. Die geplante ANSI-Norm für C sieht übrigens eine Typprüfung für ausgewählte Prozeduren (definiert als 'function prototypes') vor. In der UNIX-Welt prüfen Hilfsprogramme wie *lint* besonders Zahl und Typ der Argumente und geben notfalls Warnungen aus.

In C liefern Funktionen – wenn nichts anderes vereinbart wurde – Werte vom Typ *int*. Andere Typen müssen explizit vereinbart werden. Viele Funktionen der Bibliothek sind in den jeweiligen *#include*-Files deklariert. Der *#include*-File *math.h* enthält z.B. die Deklarationen für die mathematischen Routinen der C-Bibliothek. Es ist übrigens eine gute Praxis, Funktionen immer vor dem ersten Aufruf zu definieren, also vor dem ersten Aufruf komplett niederzuschreiben. Tun Sie dies nicht, dann nimmt der Compiler den Typ *int* an – und das kann im Einzelfall falsch sein. Als Ausweg bietet sich die sogenannte **Vorwärtsdeklaration** an. Darunter versteht man eine typgerechte Deklaration der Funktion ohne Angabe der formalen Parameter und ohne Auflistung des Funktionskörpers. Es gibt in C kein *forward*-Schlüsselwort wie in einigen Pascal-Dialekten.

Hier zwei Varianten des gleichen Programms: zuerst mit Vorwärtsdeklaration und dann mit Einhaltung der Reihenfolge Definition – Aufruf:

```
/* Variante mit Vorwaertsdeklaration  */
/* Definition erscheint nach Aufruf!  */
long produkt();   /* Vorwaertsdeklaration */

main()
{
  printf("\nProdukt aus 1000 = %lx",produkt(10));
}   /* end main */

long produkt(zahl)
int zahl;
{
  return((long)zahl * (long)zahl);
}   /* end produkt */


/* Variante mit Reihenfolge Definition - Aufruf */
long produkt(zahl)
int zahl;
{
  return((long)zahl * (long)zahl);
}   /* end produkt */
```

```
main()
{
  printf("\nProdukt aus 1000 = %lx",produkt(1000));
}   /* end main */
```

Bitte beachten Sie den Unterschied zwischen 'Deklaration' und 'Definition' einer Funktion. In der Deklaration wird nur der Typ der Funktion festegelegt – nicht der Funktionskörper, also das Programmstück, das durchlaufen werden soll. Dies geschieht erst in der Definition der Funktion, die selbst wieder eine Deklaration des Typs enthält.

10.2 Argumentübergabe

Was jetzt kommt, interessiert normalerweise keinen Anwender einer höheren Programmiersprache. Da in C allerdings 'höhere' (sprich maschinenunabhängige) Probleme wie 'niedere' (sprich systemnahe) Probleme formuliert werden können, sollten Sie zumindest grob die Details der Argumentübergabe in C kennen. Wenn Sie einmal Unterprogramme einbinden müssen, die in Assembler oder in einer höheren Programmiersprache geschrieben sind, dann werden Sie spätestens die nachfolgenden Informationen brauchen. Es tut jedoch auch unabhängig davon gut, sich mit diesen technischen Details zu beschäftigen. C ist nunmal nichts für Generalisten.

C übergibt an die gerufene Prozedur immer den Wert von Argumenten – nicht ihre Adresse. Dieser Übergabemechanismus wird **call by value** im Gegensatz zu **call by reference** (auch **call by address**) genannt. Die Eigenschaft, den Wert und nicht die Adresse zu übergeben, ist ein Charakteristikum von C. Andere Programmiersprachen bevorzugen eher die Übergabe einer Adresse. Die Argumentübergabe geschieht außerdem immer über einen **Stack** – auch das ist nicht selbstverständlich. Wenn Sie bereits Assembler programmiert haben, kennen Sie vielleicht auch die Methode, Argumente über Maschinenregister oder über besondere Parameterleisten zu übergeben. Der Rückgabewert einer Funktion wird meist über ein Maschinenregister übergeben. Schauen Sie sich die Argumentübergabe in dem folgenden einfachen Beispiel an:

```
/* proc1 ruft proc2 auf */
proc2(x,y,z)
int x,y,z;
{
int *iadr;
  iadr  = &x;
  *iadr = x * y;  /* ziemlich sinnlos */
}   /* end proc2 */
```

```
proc1()
{
int var1,var2;
  var1 = 1;
  var2 = 2;
  proc2(17,var1,var2);    /* Aufruf */
}   /* end proc1 */
```

Wenn *proc2()* durchlaufen wird (wenn der Aufruf schon erfolgt ist), hat der Stack grob (Details s. Kap. 14) das folgende Aussehen:

Wert	rel. Stackadresse
2	+8
1	+6
17	+4
Returnadresse	+2
Basepointer	+0

Wie Sie sehen, hat der Compiler nur die Werte der Argumente auf den Stack gestellt, nicht deren Adressen (die Konstante 17 hat sowieso keine Adresse im Sinne von C). Diese Werte liegen nun auf dem Stack, der ein normales Stück Speicher ist. Insofern haben die Argumente aus der Sicht von *proc2()* wieder eine Adresse! *proc2()* kann die Adresse des formalen Arguments x (entspricht dem aktuellen Argument 17) ermitteln und dort etwas abspeichern, wie im Beispiel demonstriert. Nichts Schlimmes passiert, da die 17 ja auf den Stack kopiert wurde. *call by adress* läge vor, wenn die Adressen von 17, von *var1* und *var2* als aktuelle Parameter übergeben würde, was in C etwas schwierig ist: Konstanten (außer Strings) haben keine Adresse in C.

Bei der Übergabe von Argumenten über den Stack wird temporär eine Kopie des Werts der Argumente angelegt. Deshalb sind Programme wie das folgende in C (nicht aber in Fortran) zulässig:

```
/*************************************

            Errechnet 2 hoch n
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

*************************************

Berechnet 2 hoch n

*************************************/

#include <stdio.h>

binpot(n)
int n;
{
  int wert;

  wert = 1;
  while (n-- > 0)
    wert = wert * 2;
  return(wert);
}   /* end binpot */

main()
{
  printf("\n2 hoch 5 = %d",binpot(5));
}   /* end main */
```

Dem Argument 5 im Aufruf von *binpot()* geschieht nichts Böses, obwohl das der 5 entsprechende formale Argument *n* als Laufvariable mißbraucht wird.

10.3 Schnittstellen zum Betriebssystem

Auf einige ausgewählte und häufig verwendete Funktionen muß besonders hingewiesen werden. Besonders wichtig ist die Funktion *main(),* deren Rückgabewert im allgemeinen nicht interessiert. Sie teilt dem Compiler mit, wo der handgeschriebene Teil des Programms anfängt. Sehen Sie sich ein compiliertes und gelinktes Programm mit einem Debugger an, so werden Sie nämlich feststellen, daß vor *main()* auf wundersame Art und Weise Code erscheint, der nicht von Ihnen stammt. In den meisten Fällen dient dieser Code (auch Programmvorspann oder Codepräfix genannt) dazu, die Schnittstelle zum Betriebssystem und zum Kommandointerpreter (der Kommandointerpreter akzeptiert Ihre Kommandos und führt sie aus) herzustellen. Dort werden auch Puffer und Code für die Ausführung Ihres Programms bereitgestellt. Meist endet dieser Programmvorspann mit einem Aufruf von *main().*

Eine weitere Schnittstellenfunktion Funktion ist *exit().* Sie sollte generell mit einem *int*-Wert aufgerufen werden, z.B. *exit(0)* falls das Programm korrekt beendet wurde

und *exit(1), exit(2), exit(3)* etc. im Fehlerfall. Die zulässigen Rückgabewerte von *exit()* sollten Sie Ihrem Compilerhandbuch entnehmen. *exit()* beendet Ihr Programm – notfalls mit Gewalt – und bereitet die Rückkehr ins Betriebssystem vor, wo dann der Kommandointerpreter wieder die Regie übernimmt. Dieser Kommandointerpreter erhält durch *exit()* einen Wert, der aussagt, ob das gestartete Programm korrekt oder mit Fehler beendet wurde. Wozu das gut ist? Stellen Sie sich vor, Ihr Programm läuft im Rahmen einer Kommandoprozedur ab. Dann ist es schon wichtig zu wissen, ob Ihr Programm korrekt gelaufen ist oder nicht. Die weiteren Schritte der Kommandoprozedur bauen vielleicht darauf auf, daß das Kommando Erfolg hatte.

Die C-Bibliothek (s. Kap. 12) enthält weitere wichtige vordefinierte Funktionen. Es muß daran erinnert werden, daß dort insbesondere alle Ein- und Ausgabefunktionen (*printf(), puts()* etc.) definiert sind. Die eigentliche Sprache C enthält nämlich keine Sprachelemente (Anweisungen, Operatoren) für die Ein- und Ausgabe.

10.4 Fortgeschrittene Anwendungen von Prozeduren

Die folgenden Beispiele und Kommentare zum Einsatz von Prozeduren in C-Programmen sind für Leser gedacht, die den üblichen – etwa aus Pascal bekannten – Rahmen der Programmierung verlassen wollen. Falls Sie sich nur einen ersten Überblick über C verschaffen wollen, sollten Sie das Kapitel vorläufig überschlagen. Ich behandle:

- Prozeduren mit beliebig vielen Argumenten
- rekursive Prozeduren
- Prozeduren, die Arrays oder Funktionen als Werte liefern
- Adressen von Funktionen und Pointervariablen auf Funktionen

10.4.1 Beliebig viele Argumente

Prozeduren mit beliebig vielen Argumenten sind jedem C-Programmierer bekannt. *printf()* ist eine solche Prozedur. Aber auch der Aufruf von *main(argc,argv)* deutet letztlich auf die Tatsache hin, daß *main()* beliebig viele Argumente hat.

Beginnen wir mit dem bekannten Aufruf von main():

```
/* Argumentuebergabe an main */
main(argc,argv)
int argc;
char *argv[];
```

Die Variable *argc* und der Array *argv* müssen übrigens nicht so heißen! Nahezu jeder C-Programmierer verwendet allerdings diese Namen. Diese Konstruktion dient dazu,

aus dem Kommandoprozessor des Betriebssystems Befehlsargumente an ein Programm zu übergeben. *argc* enthält die Zahl der Argumente in der Kommandozeile inklusive des Programmnamens, *argv* enthält die Adressen der einzelnen Argumentstrings. Ein Beispiel: Wird eine Kommandozeile wie

```
wsort inpfile outfile
```

angegeben, so hat *argc* den Wert 3. *argv[0]* zeigt auf den String "wsort", *argv[1]* auf den String "inpfile" und *argv[2]* auf den String "outfile". Die erste Methode der Übergabe beliebig vieler Argumente besteht also darin, einen Zähler für die Argumente und einen Array mit den Adressen der Argumente zu übergeben. Letztlich werden dann doch nur zwei Argumente übergeben! Statt der Deklaration *char *argv[]* wird bisweilen auch die Notation *char **argv* (Pointer auf Pointer) benutzt, die jedoch intuitiv nicht leicht verständlich ist.

Das folgende Programm zeigt, wie Sie die an *main()* übergebenen Argumente auswerten können. Es muß darauf hingewiesen werden, daß dieser Mechanismus nicht in C vorgeschrieben ist. Allerdings bietet C innerhalb von UNIX diese Möglichkeit, so daß sich alle ernsthaften Compiler daran orientieren. Üblicherweise werden Optionen zu Kommandos in UNIX mit einem Bindestrich '-' eingeleitet. Dieses Programm sucht danach und rettet dahinter erscheinende Parameter in globalen Variablen. Bitte beachten Sie, daß der Schleifenzähler mit dem Wert 1 beginnt, d.h. der Kommandoname selbst wird nicht geprüft.

```
/****************************************

        Testet die Argumentenuebergabe
            last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
        (C) 1987  by Markt & Technik

****************************************

Testet die Uebergabe von Argumenten an main()
Aufruf: kommando [-tm #] [-o ofile]

****************************************/

#include <stdio.h>

int testmode;
char ofilname[80];

main(argc,argv)
int argc;
char *argv[];
{
  int n;
```

```
  printf("\nAnzahl Argumente= %d",argc);
  for (n = 1; n < argc; n++)
    {
     if (strcmp(argv[n],"-tm") == 0)
       {
        sscanf(argv[n+1],"%d",&testmode);
        printf("\ntestmode= %d",testmode);
       }
     if (strcmp(argv[n],"-o") == 0)
       {
        strcpy(ofilname,argv[n+1]);
        printf("\nofilname= %s",ofilname);
       }
    }
}   /* end main */
```

Das Beispiel *printf()* geht anders vor. *printf()* muß nur ein Argument haben, den Formatstring (Details s. Kap. 12). Wenn nun in diesem Formatstring eine Konvertierungsanweisung wie *%d, %s, %lx* etc. erscheint, dann wird das jeweils nächste Argument rechts vom Formatstring ausgegeben. *printf()* hat also mindestens soviele Argumente, wie Konvertieranweisungen im Formatstring enthalten sind.

Wie können Sie nun intern auf diese Argumente zugreifen? In der Funktionsdefinition können sie nicht erscheinen, da die genaue Zahl und der Typ nicht bekannt sind. Um das zu erklären, will ich nochmals zeigen, wie die Argumente auf einer hypothetischen Maschine übergeben werden. Es wird angenommen, daß die Argumente über den Stack übergeben werden; diese Annahme trifft auf alle C-Compiler zu. Allerdings müssen Sie beachten, daß der Übergabemechanismus etwas komplizierter als in diesem Beispiel sein kann. Nehmen wir an, *printf()* werde so aufgerufen:

```
printf("Ergebnisse x1 = %d  x2 = %d",x1,x2);
```

Als einziges Argument liegt in der Prozedurdefinition die Adresse des Formatstrings vor. Also kann ich mir die Adresse dieses Strings auf dem Stack besorgen und durch Addition mit dem *sizeof*-Operator auf das nächste Argument übergehen. Im Falle von *printf()* gibt der Formatstring an, wieviele Argumente zu erwarten sind. Es gibt auch andere Möglichkeiten, dem gerufenen Programm die Zahl der Argumente mitzuteilen, z.B. durch einen definierten Wert des letzten Arguments (etwa *NULL* bei Pointerargumenten). Das folgende Beispiel zeigt stark vereinfacht, wie ein eigenes *printf()* seine Argumente erreichen könnte:

```
/****************************************

          Eigene Printf-Version
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Zugriff auf beliebig viele Argumente a la printf
myprintf.c: vereinfachtes printf: nur %d erlaubt

****************************************/

#include <stdio.h>

myprintf(format)
char *format;
{
  char *nxarg;

  /* Adresse des Pointers auf den Formatstring ermitteln */
  nxarg   =  &format;
  /* hinter diesen Pointer auf das erste Argument gehen */
  nxarg   += sizeof(char *);
  /* Formatstring durchsuchen */
  while (*format != 0)
    {
     if (*format == '%')
       {
        /* Konvertieranweisung gefunden */
        format += 2;
        printf("%d",*(int *)nxarg);  /* etwas paradox */
        nxarg += sizeof(int);
       }
     else putchar(*format++);
    }
}   /* end myprintf */

main()
{
  myprintf("Ergebnisse a1= %d  a2= %d",17,25);
}   /* end main */
```

Der Gebrauch von *sizeof* macht das Programm übrigens unabhängig von der jeweiligen Maschine (CPU). Im Unterprogramm *myprintf()* wird paradoxerweise *printf()* verwendet. Das soll jedoch nicht weiter stören; schließlich ist nicht die Zahlenkonvertierung hier mein Anliegen. Wenn Sie dort *printf()* durch eine eigene Konvertierroutine ersetzen und das Programm etwas erweitern, erhalten Sie eine ernsthafte eigene Version von *printf()*. Sie ist dann von Vorteil, wenn Sie im Grafikmodus Zeichen ausgeben müssen – dann funktioniert *printf()* meist nicht mehr. Eine solche Routine kann auch dann interessant sein, wenn Sie den ziemlich großen Platzbedarf der *printf()*-Routine für ROM-residente Programme vermeiden wollen und wenn Sie

nicht alle in *printf()* erlaubten Konvertieranweisungen ausnutzen wollen. Hier sollte das Beispiel nur den Einsatz von Prozeduren mit beliebig vielen Argumenten erläutern. Das Programm kann auch als Demonstration für die Vielseitigkeit des Pointerkonzepts in C dienen. Es soll keinesfalls dazu ermutigen, bei jeder Gelegenheit Prozeduren mit beliebig vielen Argumenten zu verwenden.

10.4.2 Rekursive Prozeduren

Rekursive Prozeduren stehen im Ruf, kompliziert und wenig effizient zu sein. Eine Prozedur ist dann rekursiv, wenn sie sich selbst direkt oder auf dem Umweg über andere (dann dynamisch eingebettete) Prozeduren aufruft. Sie taugen keinesfalls als Ersatz für alle iterative Verfahren, wie es in einigen Lisp-Lehrbüchern empfohlen wird. Im Normalfall werden Sie rekursive Prozeduren sicher selten benötigen. Die wenigen bekannten Beispiele für rekursive Prozeduren – besonders die Berechnung der Fakultät – sind rein akademische und zudem triviale Fälle. In der Praxis treten viel eher Beispiele wie das unten angeführte auf: Suchoperationen in Bäumen. Da diese rekursiven Prozeduren nicht mehr ganz so einfach sind, soll die Behandlung von Variablen und Argumenten in C noch einmal erläutert werden.

Für rekursive Prozeduren spielt der Stack eine große Rolle. Lokale (interne) Prozeduren gibt es in C nicht. Damit werden Sie also keinen Ärger haben. Formale Argumente werden auf dem Stack angelegt (wenn sie nicht in Registern abgelegt werden sollen). Sie leben damit nur solange, wie die dazugehörige Prozedur durchlaufen wird. Für interne Variablen gilt das gleiche, sofern sie nicht *static* deklariert sind. Bei jedem rekursiven Aufruf einer Prozedur entsteht ein **Stackframe**, eine Menge von Informationen auf dem Stack aus internen Variablen, Prozedurargumenten, Rückkehradressen und geretteten Registern. Ein Stackframe sieht etwa so aus, wie im obigen Beispiel für Prozeduren mit beliebig vielen Variablen angegeben. Sie können sich vorstellen, daß bei sehr tiefer Rekursionsschachtelung Probleme entstehen: es wird immer mehr Platz auf dem Stack beansprucht – der Stack kann überlaufen ('stack overflow'). Erst bei einem *return* aus einer rekursiven Prozedur wird ein Stackframe freigegeben, d.h. der Stack wird kleiner. Jedem Aufruf einer rekursiven Prozedur entspricht ein solcher Stackframe eindeutig. Der Effekt? Gleichnamige Argumente und interne Variablen können unterschiedliche Werte im Verlauf der Aufrufgeschichte aufweisen. Ein *return* aus einem rekursiven Aufruf liefert wieder die Werte für Argumente und lokale Variablen, die vor dem Aufruf gegolten haben! Bitte beachten Sie: das gilt nicht für statische Variablen. Sie bleiben von der ganzen Rekursion unberührt. Interne statische Variablen und externe Variablen (beide werden im Datensegment, nicht im Stacksegment abgelegt) können also dazu verwendet werden, Zwischenergebnisse vor der Rekursion zu schützen.

In den C-Compilern brauchen Sie nicht (wie in einigen Pascal-Compilern) die jeweilige Prozedur extra mit einer Compileroption 'rekursiv' zu übersetzen. Die Fähigkeit zur Rekursion ist eine Standardeigenschaft von C-Prozeduren.

Das folgende Beispielsprogramm stellt eine baumorientierte Sortierroutine ('tree sort') dar. Es liest Wörter ein, erstellt einen Sortierbaum, an dessen Knoten die Wörter als Knotenetiketten stehen (Prozedur *mache_knoten()*) und gibt den Sortierbaum wieder aus (Prozedur *drucke_knoten()*). Beide erwähnten Routinen sind rekursiv. Wenn Sie sich näher mit rekursiven Prozeduren beschäftigen wollen, sehen Sie sich zuerst die Routine *drucke_knoten()* an; sie ist besonders einfach. Die Knoten des Sortierbaumes sind durch die selbstbezügliche Struktur *knoten* repräsentiert. Gleiche Wörter werden übrigens nur einmal aufgenommen; die Häufigkeit der Wörter wird mitgezählt:

```
/***************************************

              Sortierroutine
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

***************************************

Sortieren ueber einen Baum
Beispiel fuer rekursive Prozeduren

***************************************/

#include <stdio.h>
#include <ctype.h>

/* Aufbau eines Baumknotens */
struct knoten
{
  char *word;
  int haeuf;              /* Haeufigkeit */
  struct knoten *links;   /* linker Nachfolger */
  struct knoten *rechts;  /* rechter Nachfolger */
};

/* legt einen Knoten mit Etikett w an */
struct knoten *mache_knoten(p,w)
struct knoten *p;     /* Vorgaenger */
char *w;              /* Wort (Etikett) */
{
  int cond;

  if (p == NULL)
    {
     /* Knoten ist neu */
     p = malloc(sizeof(struct knoten));
     p->word = malloc(strlen(w) + 1);
     if (p->word != NULL) strcpy(p->word,w);
     p->haeuf = 1;
     p->links = p->rechts = NULL;
    }
```

```
  else
    if ((cond = strcmp(w,p->word)) == 0)
            p->haeuf++;                 /* Wort gibts schon */
    else if (cond < 0)                  /* links einhaengen */
            p->links = mache_knoten(p->links,w);
    else                                /* rechts einhaengen */
            p->rechts = mache_knoten(p->rechts,w);
    return(p);
}   /* end mache_knoten */

/* Baum rekursiv ausdrucken */
int drucke_knoten(p)
struct knoten *p;
{
  if (p != NULL)
    {
     drucke_knoten(p->links);
     printf("%4d %s\n",p->haeuf,p->word);
     drucke_knoten(p->rechts);
    }
}   /* end drucke_knoten  */

main()
{
  struct knoten *wurzel;
  char buffer[80];
  int t;

  wurzel = NULL;
  for (;;)
    {
     printf("Naechstes Wort: ");
     gets(buffer);
     if (strlen(buffer) == 0) break;
     t = buffer[0];
     if (isalpha(t))
       wurzel = mache_knoten(wurzel,buffer);
    }
  drucke_knoten(wurzel);
}   /* end main */
```

Die folgende Zeichnung zeigt einen Sortierbaum nach obigem Muster, in dem die Namen 'Hans', 'Arnold', 'Abraham' und 'Peter' abgelegt wurden. Bitte versuchen Sie, die Arbeitsweise der beiden rekursiven Prozeduren anhand dieses Baums zu verstehen:

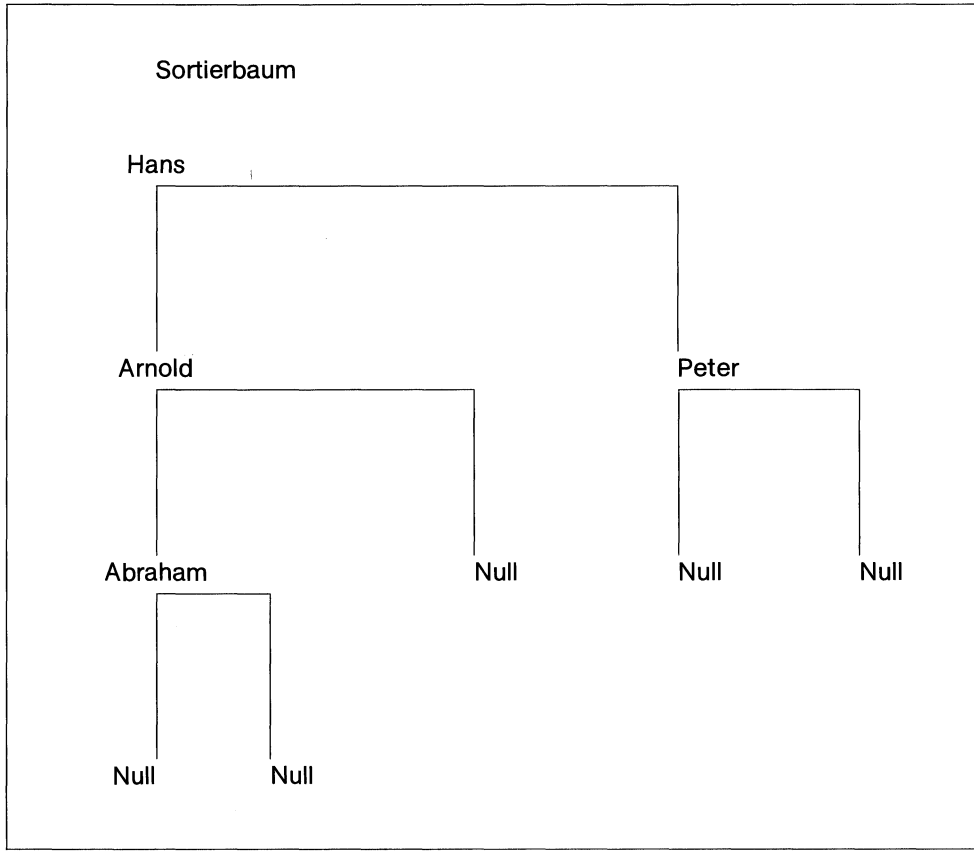


Bild 10.1: *Sortierbaum*

10.4.3 Funktionen mit besonderen Rückgabewerten

Funktionen in C können nur einfache *(int, long, ...)* oder abgeleitete *(char *, int *, ...)* Werttypen liefern. Was, wenn Sie einen Array oder eine Struktur als Ergebnis einer Funktion brauchen? Keine Angst: Sie brauchen nicht auf die neue ANSI-Norm für C zu warten – es geht auch ohne.

Wenn Funktionen schon nicht Werte mit zusammengesetzten Typen (Arrays, Strukturen) liefern können, dann doch Pointer auf solche Objekte: Pointer auf Arrays bzw. Strukturen sind nämlich wieder abgeleitete Typen. Natürlich gibt es auch die Möglichkeit, Funktionen gar nicht als Funktionen, sondern als normale Prozeduren zu schreiben, die ihre Ergebnisse irgendwo in globalen Variablen (als Seiteneffekt) hinterlassen. Das jedoch gilt als unfein. Das folgende Programm zeigt eine typische Alternative. Die Prozedur erhält als Argument einen Promptstring (eine Eingabeaufforderung) und liefert als Ergebnis einen Zeiger auf die gewünschte Eingabe:

```
/*************************************

            Zeigerermittlung
          last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

*************************************

Dieses Programm liefert den Zeiger auf ein static-Array

*************************************/

#include<stdio.h>

char *myinput(prompt)
char *prompt;
{
  static char outp[100];

  printf("\n%s",prompt);
  gets(outp);
  return(outp);
}   /* end myinput */

main()
{
  char *hadr;

  hadr = myinput("Bitte ein Wort eingeben:");
  printf("\nErgebnis: %s",hadr);
}   /* end main */
```

Bitte beachten Sie – das ist hier sehr wichtig – daß in der Prozedur *myinput()* der Ablagearray *outp* als *static* deklariert wurde. Er darf keinesfalls auf dem Stack angelegt werden, weil sonst ein folgender Aufruf auf irgendeine andere Prozedur den gelesenen Inputstring zerstören würde. Der gelieferte Funktionswert ist ein Zeiger auf den Anfang des Strings *outp*. Das gleiche Verfahren läßt sich auf Strukturen anwenden. Die Struktur sollte innerhalb der gerufenen Prozedur mit dem Attribut *static* deklariert werden. Als Funktionswert wird ein Pointer auf die Struktur geliefert.

10.4.4 Pointer auf Funktionen

Pointer auf Funktionen (Funktionsvariablen) sind wirklich eine Eigenschaft der Sprache C, die Sie kaum in anderen Programmiersprachen außer in Assembler finden. Pointer auf Funktionen sind in C deshalb konzipierbar, weil der Funktionsaufruf () ein C-Operator ist, der auf Pointer angewendet werden kann – ob Konstante oder Variable ist gleichgültig.

Die Deklaration einer Funktionsvariablen erfordert zwingend runde Gruppierungsklammern um den Funktionsnamen und den Pointerstern wie in den folgenden Beispielen:

```
/* Deklaration von Funktionsvariablen */
int (*pfint)();
char (*pfpchar)();
long (*pflong)();
```

Der Sinn von Funktionsvariablen liegt – das hat zumindest meine praktische Erfahrung ergeben – vor allem im Einsatz bei der Verarbeitung von Tabellen, wie z.B. von Tastaturtabellen in Editoren oder von Funktionsauswahltabellen in Terminalemulationen.

10.5 Zusammenfassung

In C sind alle Prozeduren Funktionen, d.h. alle Prozeduren liefern Werte. Man unterscheidet zwischen formalen und aktuellen Argumenten einer Prozedur. C zwingt nicht zum Einhalten des Typs der Funktion noch der Argumente. Die gute Programmierpraxis achtet jedoch auf die Einhaltung der Typen. Funktionen können nur einfache oder abgeleitete Werttypen liefern. Es gibt jedoch Auswege aus dieser Restriktion. Der grundlegende Übergabemechanismus ist 'call by value'. 'call by reference' ist möglich. In C sind Funktionsvariablen erlaubt, die in tabellengesteuerten Funktionsverteilern von großem Vorteil sind.

11 Der C-Präprozessor

11.1 Funktionen des Präprozessors

Jeder C-Compiler besteht im wesentlichen aus drei Teilen: dem Präprozessor, dem Parser (Syntaxprüfer) und dem Codegenerator. Der Präprozessor ist – wie der Name sagt – ein 'Vor'-Prozessor, also ein Programm, das den Quellcode wie ein Filter analysiert und daraus einen neuen Quellcode erzeugt. Im allgemeinen besteht die Filterung (und das widerspricht etwas dem üblichen Sprachgebrauch) aus einer Erweiterung (Expandierung) des Quellcodes. Erst der so gefilterte Quellcode wird dem eigentlichen Compiler vorgesetzt. Wie man sieht, ist der Präprozessor kein lebensnotwendiger Teil des jeweiligen Compilers. Man kann deshalb auch Compiler finden, bei denen der Präprozessor ein eigenständiges, abgetrenntes Programm ist. Solche Präprozessoren (natürlich selbst in C geschrieben) sind auch als Public-Domain-Programme zu finden.

Es können Fehler durch die Expansion des Quellcodes auftreten, die im komprimierten Quellcodeoriginal nur sehr schwer erkennbar sind. Diese Fehler sind bisweilen so heimtückisch, daß ich es mir angewöhnt habe, den Präprozessor nur sehr sparsam einzusetzen und bewußt nur die einfacheren Mechanismen zu verwenden. Insgesamt gesehen aber ist der Präprozessor ein wichtiges Hilfsmittel in der C-Programmierung.

Was hat dieser Präprozessor nun für eine Funktion? In der Praxis haben sich die folgenden Vorteile beim Einsatz des Präprozessors erwiesen:

- die C-Programme werden weitgehend unabhängig von der jeweiligen Hardwareumgebung.
- die C-Programme werden weitgehend unabhängig vom jeweiligen Betriebssystem.

- die C-Programme können an verschiedene Endbenutzeranforderungen angeglichen werden.
- immer wiederkehrende Quellcodesequenzen können als Makros formuliert werden.

Präprozessorkommandos beginnen mit dem Zeichen # in Spalte 1 einer Schreibzeile; in einigen Implementierungen reicht es auch, wenn das # das erste Zeichen in der Zeile ist, das nicht einem Weißraum entspricht. Da der Präprozessor nicht Bestandteil der Sprache C ist, gelten hier nicht die flexiblen Schreibregeln von C. Falls ein Präprozessorkommando einmal länger als eine Zeile werden muß, kann ein Backslash unmittelbar gefolgt von einem Newline-Zeichen (Carriage Return bei der Eingabe) als Fortsetzungskennzeichen dienen. Präprozessorkommandos werden im Allgemeinen nicht mit einem Semikolon abgeschlossen – es liegen ja keine Befehle der Sprache C vor.

11.2 Definieren von Konstanten und Makros

Der Präprozessorbefehl *#define* erlaubt es, feste Strings im Quellcode durch andere Strings zu ersetzen. Üblicherweise werden mit *#define* Konstanten und Makros definiert; es ist jedoch auch möglich, Bestandteile (Schlüsselwörter, Separatoren, Operatoren) der Sprache C umzudefinieren. Beginnen wir mit einem einfachen Beispiel. Auf den meisten Maschinen gilt LF (Code 0x0a) als Zeilenendzeichen. Auf einigen Maschinen jedoch ist in ASCII das vorderste Bit gesetzt (Code 0x8a). Auf anderen Maschinen wiederum gilt Carriage Return als Zeilenendzeichen (Code 0x0d). Also werde ich für die meisten Maschinen den Code für Linefeed so definieren:

```
#define LF 0x0a
```

Für einige Maschinen muß der Code dann so definiert werden:

```
#define LF 0x8a
```

Natürlich können auch dezimale oder oktale Werte hier angegeben werden, also z.B.

```
#define LF 10
#define LF 012
#define LF '\n'
#define LF '\012'
```

Auf der rechten Seite von *#define*-Befehlen ist so ziemlich alles erlaubt, was auch in C erlaubt ist. Auf einen häufigen Fehler sollten Sie achten: zwischen dem definierten Symbol und seinem Wert darf kein Gleichheitszeichen stehen. Der Präprozessor wird Ihnen das Gleichheitszeichen ungerührt in den Quellcode eintragen. An *#define* ist nichts Geheimnisvolles: alle Vorkommen des ersten Symbols hinter *#define* werden vom Präprozessor durch die Zeichen rechts von diesem Symbol ersetzt. Wenn eine

Zeile für die Definition zu kurz ist, kann mit '\' gefolgt von einem Linefeed eine Fortsetzungszeile angekündigt werden.

Durch *#define* können auch Makros definiert werden. Makros sind Codefolgen, die vom Präprozessor in den modifizierten Quellcode eingefügt werden. Ein Makro ist an der runden öffnenden Klammer erkennbar, die unmittelbar auf das definierte Symbol folgt. Wie Prozeduren können Makros mit Parametern versehen werden. Während der Körper einer Prozedur (der durchlaufene Code) nur einmal in einem Programm erscheint, taucht der durch ein Makro definierte Code sooft in einem Programm auf, wie das Makro genannt wird (ich vermeide absichtlich den Ausdruck 'aufgerufen'). Die Ersetzung der formalen durch die aktuellen Parameter erfolgt im Falle eines Makros schon vor der Compilezeit; der vom Präprozessor expandierte Quellcode weist nur noch die aktuellen Parameter auf. Makros haben gegenüber Prozeduren den Vorteil der geringeren Ausführungszeit: es findet ja keine Parameterübergabe zur Laufzeit statt, kein Call-Befehl muß ausgeführt wrden (wenn nicht das Makro selbst einen Prozeduraufruf enthält). Demzufolge werden Makros häufig für Probleme eingesetzt, die schnelle Programmlaufzeiten erfordern. Sehr häufig werden sie einfach auch nur aus Bequemlichkeit definiert: der Programmierer ist es einfach leid, immer die gleiche Codesequenz hinzuschreiben. Makros haben den Nachteil, daß der erzeugte Code größer wird, da der durch das Makro definierte Quellcode ja u.U. mehrfach auftaucht. Auch hier wieder die Warnung: beim Testen eines Programms sorgen Makros für Überraschungen. Da meist nicht der expandierte Quellcode, sondern der Originalquellcode zum Debuggen herangezogen wird, sorgen Makros für Verwirrung.

Das folgende Beispiel zeigt eine Makrodefinition und dann zwei Codestücke. Das erste Codestück zeigt das Originalprogramm, wie Sie es geschrieben haben könnten. Das zweite Codestück zeigt das vom Präprozessor expandierte Codestück:

```
/* Codestueck 1: Originalcode */
#define ROUND(x) ((int)(x + 0.5))

main()
{
float xyz;
  xyz = 0.6;
  printf("\n%d",ROUND(xyz));
  printf("\n%d",ROUND(1.1));
}

/* Codestueck 2: vom Praeprozessor expandiert */
main()
{
float xyz;
```

```
  xyz = 0.6;
  printf("\n%d",((int)(xyz + 0.5)));
  printf("\n%d",((int)(1.1 + 0.5)));
}
```

Wie Sie sehen, hat der Präprozessor den formalen Parameter *x* durch die aktuellen Parameter *xyz* bzw 1.1 ersetzt. Es sollte Sie nicht verwirren, daß im expandierten Codestück 2 die Makrodefinition nicht mehr auftaucht. Der Präprozessor hat seine Arbeit getan und setzt dem Compiler ein von Präprozessorbefehlen gereinigtes Programm vor. Vielleicht finden Sie, in der Definition sei mindestens ein Klammernpaar überflüssig? Richtig, aber das ist reine Vorsicht. Gerade in Makrodefinitionen sollten Sie eher zuviel als zuwenig klammern. Schauen Sie sich das folgende Beispiel an. In der Definition des Makros wird auf die üblichen Vorrangregeln (Punktrechnung vor Strichrechnung) vertraut. Der Präprozessor produziert eine Resultat, das sicher nicht gewünscht war:

```
/* Originalcode: Makrodefinition riskant  */
#define BERECHNE(x,y,z) (x + y * z)
main()
{
  printf("\n%d",BERECHNE(5,10+2,20+3));
}

/* Nach Expansion */
main()
{
  printf("\n%d",(5 + 10+2 * 20+3));
}
```

Bei der Definition von Makros dürfen Sie sich keine Schreibfreiheiten wie bei der Definition von Prozeduren erlauben! Zwischen *ROUND* und *(x)* darf im obigen Beispiel kein Blank stehen – sonst gilt die *#define*-Zeile nicht als Makro, sondern als Definition einer Konstanten, die den üblichen Ersetzungsregeln unterliegt. Schauen Sie sich an, was der Präprozessor aus einer derart fehlerhaften Makrodefinition macht:

```
/* Makrodefinition (fehlerhaft!) */
#define BERECHNE (x,y,z) (x + y * z)
main()
{
  printf("\n%d",BERECHNE(5,10+2,20+3));
}
```

```
/* Nach Codeexpansion: boeser Fehler */
main()
{
  printf("\n%d",(x,y,z) (x + y * z)(5,10+2,20+3));
}
```

Ein gering aussehender Fehler – das Blank zwischen *BERECHNE* und der öffnenden runden Klammer muß weg – hat offensichtlich böse Folgen: der Präprozessor hat nicht mehr den Eindruck, daß er es mit einem Makro zu tun hat. Er ersetzt *BERECHNE* ganz einfach durch den Ausdruck, der rechts vom folgenden Blank steht. Beim Compilieren dieses Beispiels erhalten Sie übrigens eine Fehlermeldung, die Sie möglicherweise verwirrt, weil Sie den expandierten Code wahrscheinlich nicht vor Augen haben.

Schließlich gibt es noch das Problem der unerwünschten **Seiteneffekte**, für die Makros besonders anfällig sind. Im Kapitel über die C-Bibliothek wird öfters darauf hingewiesen. Beim Schreiben eines Makronamens im Quelltext hat man selten die genaue Makrodefinition im Kopf. Wenn dann noch mit Inkrement- oder Dekrementoperatoren hantiert wird, gibt es leicht unerwünschte Effekte wie im folgenden Beispiel:

```
/* Eine moegliche Definition von toupper */
#define toupper(c) (((x >= 'a') && (x <= 'z')) ? \
                        (x - ' ') : x)
```

Diese Definition ist eigentlich für den ASCII-Bereich korrekt und könnte auch so in einer Prozedur erscheinen; in einem Makro allerdings richtet Sie großes Unheil an, wenn das Makro gedankenlos verwendet wird. Den folgenden Output des Präprozessors habe ich erhalten, nachdem ich das Makro so verwendet habe:

```
char *hadr;
...
  if (toupper(*hadr++) == 'A') return;


/* Output des Praeprozessors fuer dieses */
/* Codestueck                            */
  if ((((*hadr++ >= 'a') && (*hadr++ <= 'z'))
        ? (*hadr++ - ' ') : *hadr++)  == 'A')
return;
```

Wie Sie sehen, ist das durchaus keine exotische Verwendung des Makros – und dennoch wird sicherlich nicht gewünschter Code produziert. *hadr* wird nämlich dreimal inkrementiert statt einmal, wie wohl beabsichtigt! Bei Verwendung einer Prozedur wird dieses Problem nicht auftreten, da der Wert von *hadr* zuerst als Argument auf den Stack gestellt wird und dann *hadr* inkrementiert wird. In der Prozedur kann kein

großes Unheil mit *hadr* geschehen, da durch den 'call by value' Mechanismus nicht mit der Originalvariablen *hadr* gerechnet wird, sondern mit der Kopie auf dem Stack.

In der professionellen Programmierpraxis hat es sich bewährt, alle vom Präprozessor übersetzten Konstanten und Makros mit Großbuchstaben zu schreiben. Beim Lesen erkennen Sie leicht die Ausdrücke in einem Programm, die vom Präprozessor ersetzt werden. Vielleicht sind Sie dann – besonders wenn es um Makros geht – nicht mehr überrascht, beim Testen und Debuggen das fragliche Symbol nicht mehr wiederzufinden. Es ist ja ersetzt worden: der Compiler findet nicht Ihren Originalquellcode vor!

Halten wir also fest: Benutzen Sie Makros nur, wenn Sie sich sicher sind, alle Ersetzungsmechanismen und alle Restriktionen wirklich verstanden zu haben. Setzen Sie Makros sparsam ein.

11.3 Einkopieren von Quellcode

In den sehr einfachen Beispielprogrammen vom Anfang dieses Buches sind uns schon Präprozessorkommandos begegnet: *#include<stdio.h>* ist ein solches Präprozessorkommando, das das Einkopieren der Definitionsdatei *stdio.h* in den Quellcode bewirkt. Diese Zeile kommt in den meisten C-Programmen vor. Sie dient dazu, besonders häufige Konstanten (*NULL, EOF*) zu definieren, den Typ wichtiger Bibliotheksroutinen zu deklarieren und vor allem das zeichenorientierte IO der mit 'f' beginnenden Standardroutinen auf die Basisroutinen abzubilden. Da diese Zeile

```
#include<stdio.h>
```

ein besonders häufig gebrauchtes Kommando ist, will ich einige Details der dazugehörigen Datei *stdio.h* erläutern:

```
/* ------- Auszug aus einem stdio.h --------- */
#define  BUFSIZ  512
#define  _NFILE  20
#define  FILE    struct _iobuf
#define  EOF     (-1)
extern FILE
{
    char *_ptr;
    int   _cnt;
    char *_base;
    char  _flag;
    char  _file;
} _iob[_NFILE];

#define  stdin   (&_iob[0])
#define  stdout  (&_iob[1])
```

```
#define  stderr  (&_iob[2])
#define  stdaux  (&_iob[3])
#define  stdprn  (&_iob[4])

#define  _IOREAD    0x01
#define  _IOWRT     0x02
#define  _IONBF     0x04
#define  _IOMYBUF   0x08
#define  _IOEOF     0x10
#define  _IOERR     0x20
#define  _IOSTRG    0x40
#define  _IORW      0x80

#define  getc(f)    (--(f)->_cnt >= 0 ? 0xff &  \
                    *(f)->_ptr++ : _filbuf(f)
#define  putc(c,f)  (--(f)->_cnt >= 0 ? 0xff & \
                    (*(f)->_ptr++ = (c)) : \
                    _flsbuf((c),(f)))

#define  getchar()   getc(stdin)
#define  putchar(c)  putc((c),stdout)

#define  feof(f)     ((f)->_flag & _IOEOF)
#define  ferror(f)   ((f)->_flag & _IOERR)
#define  fileno(f)   ((f)->_file)
```

In dem Auszug aus einem typischen *stdio.h* werden zunächst einige Konstanten, u.a. die Puffergröße *BUFSIZ* und die wichtige Konstante *EOF* definiert. Es wird (durch *_NFILE*) festgelegt, daß höchstens 20 Files in den Standardroutinen (das sind die Routinen, die Filepointer statt Dateinummern verwenden) gleichzeitig verwendet werden können. Der wichtige Datentyp *FILE* wird als Synonym für das Strukturmuster *_iobuf* definiert. Sie erkennen anhand der Struktur *FILE* alias *_iobuf*, daß dort kein Puffer direkt zur Verfügung gestellt wird, sondern nur ein Pointer auf diesen Puffer. Der Puffer selbst wird entweder beim Starten des Programms (für *stdin, stdout* etc.) automatisch allokiert oder beim expliziten Öffnen eines Files. Die standardmäßig immer vorhandenen Filepointer *stdin* (Standardeingabe), *stdout* (Standardausgabe) und *stderr* (Fehlerausgabe) werden definiert als Adressen der ersten drei Elemente des Arrays *_iob*; sie werden also wie normale Files behandelt. Schließlich werden noch einige Makros für die Ein-/Ausgabe definiert, u.a. die zeichenorientierten Routinen *getchar()* und *putchar()* für den Terminalbetrieb.

Ähnliche aber nicht notwendigerweise identische Versionen von *stdio.h* finden sich im Umfang jedes C-Compilers. Jeder Compiler wird aber leicht unterschiedliche Versionen aufweisen. Wie man schon an diesem Auszug sieht, können in *#include*-Dateien wiederum Präprozessor-Kommandos und sogar *#include*-Kommandos

erscheinen. Aber Vorsicht: Auch Endlosschleifen sind möglich! Wenn man in einer einkopierten Datei wiederum die gleiche Datei oder eine höher liegende Datei aufruft, erlebt man die schönsten Überraschungen.

Mit *#include*-Befehlen können sie nicht nur wie im gerade gezeigten Beispiel Typdefinitionen, Makros und Konstanten in den Quellcode einkopieren, es steht Ihnen auch frei, komplette Programme in ein anderes Programm zu 'importieren'. Sie können dies an beliebiger Stelle Ihres Programms tun. Der *#include*-Befehl muß nicht am Anfang Ihres Programms stehen.

Vielleicht ist Ihnen schon einmal aufgefallen, daß die meisten *#include*-Befehle den nachfolgenden Filenamen in spitzen Klammern einkleiden (wie im obigen Beispiel), einige jedoch die Fileangabe in doppelte Anführungen aufweisen. Für den gleichen Filenamen sind also die beiden Varianten möglich:

```
#include <xyz.h>
#include "xyz.h"
```

Im ersten Fall wird zuerst ein spezielles Directory durchsucht und dann erst das aktuelle Directory. Im zweiten Fall ist die Suchreihenfolge umgekehrt: es wird zuerst auf dem aktuell eingestellten Directory nach dem File *xyz.h* gesucht. Das ist ganz nützlich, wenn 'private' Versionen von Standard #include-Files existieren, die nicht im 'Systempool' abgelegt sind, aber noch den alten Namen tragen, z.B. modifizierte Versionen von *stdio.h*. Die Schreibweise mit doppeltem Anführungszeichen wird man auch wählen, wenn der einzukopierende File nur Informationen enthält, die nicht systemweit interessant sind.

Hier ist eine unvollständige Übersicht der wichtigsten *#include*-Files. Sie werden feststellen, daß nicht alle C-Compiler über die gleichen #include-Files verfügen:

- stdio.h: enthält die Strukturen für die Standardroutinen zur Ein-/Ausgabe, enthält die Definitionen der Konstanten *EOF* und *NULL*.
- math.h: enthält die Funktionsdeklarationen für die mathematischen Funktionen der C-Bibliothek wie *sqrt()* etc.
- ctype.h: enthält die Prozedur- und Makrodeklarationen für die Konvertierungsroutinen wie *toupper()* etc.

11.4 Bedingtes Kompilieren

Unter dem Begriff 'Bedingtes Kompilieren' (conditional compiling) versteht man das Unterfangen, das gleiche Quellprogramm an unterschiedliche Betriebssysteme, Compiler oder Hardwareumgebungen anzupassen. Dies wird dadurch erreicht, daß bestimmte Codestrecken bei Setzen einer Konstanten ausgelassen bzw. kompiliert werden. Im folgenden Beispiel wird dies anhand zweier C-Compiler verdeutlicht: Der

an UNIX V7 orientierte DRC-Compiler kennt die Funktionen *index(), opena()* für ASCII-Dateien und *openb()* für Binärdateien. Er kennt ferner den Typ *unsigned char* nicht. Der MSC-Compiler orientiert sich eher an XENIX und kennt die Funktionen *strchr()* (statt *index()*) und nur eine einzige *open()*-Funktion, die per Parameter auf binäre oder ASCII-Verarbeitung eingestellt werden kann. Der DRC-Compiler kennt wie viele andere Systeme auch den *#include*-File *fcntl.h* nicht, der im Falle des MSC-Compilers die Konstanten *O_BINARY* und *O_TEXT* definiert. Im Programm soll nur der Typ *CHAR* und die Funktionsnamen *index, aopen* und *bopen* verwendet werden. Das nachstehende Codestück nimmt an, daß der DRC-Compiler eingesetzt werden soll. Hier werden nur Definitionen und Makros bedingt definiert. Natürlich können auch echte C-Programmteile zwischen *#ifdef* und *#else* und *#endif* eingesetzt werden.

```
/* Version 1 */
#define DRC
#ifdef DRC
#define CHAR char
#define aopen(x,y) opena(x,y)
#define bopen(x,y) openb(x,y)
#else
#include <fcntl.h>
#define CHAR unsigned char
#define aopen(x,y) open(x,y+O_TEXT)
#define bopen(x,y) open(x,y+O_BINARY)
#endif
```

Die Codestrecke nach *#ifdef DRC* wird nur dann ausgeführt, wenn die Konstante *DRC* irgendwie definiert wurde – das ist hier der Fall. Welchen Wert diese Konstante hat, spielt keine Rolle. Die durch *#ifdef DRC* eingeleitete bedingt kompilierte Strecke wird durch *#endif* beendet. *#ifdef* und *#endif* bilden also eine Klammer. In unserem Beispiel wird diese Klammer durch eine *#else*-Strecke unterteilt. Zu dieser Konstruktion ist nicht viel zu sagen: sie entspricht in der Syntax einem zusammengesetzen *if-else* Block in C, ist allerdings etwas schlechter lesbar, da ich keine Einrückungen vorgenommen habe. Leider akzeptieren nicht alle Präprozessoren solche Einrückungen. *aopen* und *bopen* sind in beiden Fällen als leicht unterschiedliche Makros definiert. Die *#else*-Strecke enthält wieder eine *#include*-Anweisung. Wenn statt des DRC-Compilers irgend ein anderer Compiler verwendet werden soll, dann genügt es, den Befehl *#define DRC* zu entfernen oder ihn in Kommentare zu setzen.

Wenn das Programm wirklich nur zu den beiden genannten Compilern paßt, sollten die bedingt kompilierfähigen Strecken besser so aussehen:

```
/* Version 2 - verkuerzt */
#define DRC
#ifdef DRC
   ....
#endif
#ifdef MSC
   ....
#endif
```

Diese Version (kein #else mehr!) schließt den Fall aus, daß irgendein dritter Compiler ins Spiel kommt. Dafür ist nämlich kein Fall vorgesehen und der Compiler sollte dann einen Fehler melden, weil er die Funktionen *aopen()* und *bopen()* (dafür ist jetzt kein Makro definiert) nicht kennt.

Die negative Version von *#ifdef* ist *#ifndef* (führe aus, wenn die nachfolgende Konstante nicht definiert ist). Die nachfolgende Variation der ersten Version unterscheidet die Fälle 'Nicht-DRC-Compiler' und 'alle anderen Compiler':

```
/* Version 3 */
#define DRC
#infdef DRC
#include <fcntl.h>
#define CHAR unsigned char
#define aopen(x,y) open(x,y+O_TEXT)
#define bopen(x,y) open(x,y+O_BINARY)
#else
#define CHAR char
#define aopen(x,y) opena(x,y)
#define bopen(x,y) openb(x,y)
#endif
```

Die *#if...* und *#else*-Strecken sind jetzt vertauscht!

In einer *#ifdef*, *#ifndef* oder *#else*-Strecke kann selbst wieder eine *#ifdef* oder *#ifndef* erscheinen. Sie sehen jedoch auch hier, daß die Programme nicht gerade besser lesbar werden, wenn Schachtelungen von Präprozessorbefehlen eingesetzt werden.

Schließlich gibt es noch den *#if*-Befehl. Die Strecke danach bis zum *#else* oder zum *#endif* wird ausgeführt, wenn der Konstantenausdruck hinter *#if* einen Wert ungleich Null ergibt. Dieses Beispiel zeigt, daß die bedingte Kompilierung für das Austesten von Programmen interessant sein kann. Beispiel:

```
#define TEST 17+4
.....
.....
#if TEST
  printf("\nTestpunkt 1 erreicht");
#endif
```

11.5 Zusammenfassung

Der Präprozessor erzeugt aus dem Originalquellcolcode, der dann dem eigentlichen Compiler vorgesetzt wird. Die Präprozessorkommandos beginnen in Spalte Eins. Zugelassene Kommandos sind meist *#include* (Einkopieren von Definitionsdateien), *#define* (Definition von Konstanten und Makros), *#if* (mit Varianten, Beginn einer bedingten Kompilationsstrecke) und *#endif* (Ende einer bedingten Kompilationsstrecke). Bei der Verwendung von Makros sind einige Restriktionen zu beachten. Makros werden über *#define* definiert; sie sind wie Funktionen parametrisierbar, erfüllen jedoch nicht den gleichen Zweck. Die Verwendung von Präprozessorkommandos macht C-Programme leichter anpaßbar an verschiedene Benutzeranforderungen und an unterschiedliche Hardware- und Betriebssystemumgebungen. Es empfiehlt sich, Präprozessorkommandos sparsam einzusetzen, da sie nicht immer zur Lesbarkeit eines Programms beitragen.

Hier die wichtigsten Präprozessorkommandos:

```
#define
#include
#if
#ifdef
#ifndef
#else
#endif
```

12 Die C-Bibliothek

12.1 Einführung in die C-Bibliothek

Im nachfolgenden Kapitel werden die wichtigsten Bibliotheksfunktionen vorgestellt. Ich beschränke mich auf die Funktionen, die in nahezu allen Compilern vorhanden sind. Soweit compiler- und systemabhängige Besonderheiten mir schon einmal aufgefallen sind, sind diese hier vermerkt. Da jedoch insbesondere die IO-Funktionen nicht Bestandteil der Sprache selbst sondern der jeweiligen Bibliotheken sind, sollten Sie im Zweifelsfall in Ihrem Compilerhandbuch nachsehen oder noch besser das Verhalten der fraglichen Routinen durch kurze Testprogramme prüfen. Das Testprogramm am Ende des Kapitels könnte einen geeigneten Testrahmen bilden.

Beim Transport von Programmen sind mir selbst bei den elementarsten Routinen der C-Bibliothek Fehler und unvorhergesehene Verhaltensweisen begegnet. Der Grund liegt wohl darin, daß die Bibliotheksfunktionen in K&R nur sehr dürftig beschrieben sind und dort eher zeigen sollen, wie man solche Funktionen aus Systemaufrufen und C-Sprachmitteln schreiben kann. In der Tat können die Bibliotheksfunktionen, falls sie einmal fehlen oder fehlerhaft sind, auf den jeweiligen Systemen leicht nachgebildet werden. Voraussetzung dafür ist natürlich, daß die benutzten Konzepte überhaupt auf dem System vorhanden sind. So wird man z.B. eine Funktion wie *chdir()* (change directory) schwerlich in CP/M 2.2 nachbilden können, da es dort im Standard nur ein einziges Directory (das der kompletten Diskette) gibt. Das Schreiben eigener Bibliotheksroutinen als Ersatz vorhandener Routinen machte auf einigen Compilern (oder besser gesagt: mit den dazugehörigen Linkern) Ärger, wenn die gleichen Namen verwendet wurden. Vorsichtige Programmierer werden also leicht unterschiedliche Namen verwenden, wenn Bibliotheksroutinen durch eigene Routinen ersetzt werden.

Nahezu alle Compilerbauer bemühen sich, die Bibliotheksfunktionen am UNIX-Vorbild zu orientieren. Da es jedoch inzwischen auch mehrere UNIX-Versionen gibt, ist die Lage insbesondere auf dem Gebiet der Stringverarbeitungsroutinen verworren. So heißt z.B. die Routine *index()* auf neueren Compilern meist *stdchr()*. Das Problem der Variation der Namen von ansonsten inhaltsgleichen Routinen kann leicht durch eine Deklaration im Präprozessorteil eines Programms behoben werden.

Leute, die öfters mal mit einem symbolischen Debugger in ihre Programme schauen, werden einige Bibliotheksfunktionen weder in der Symboltabelle noch im Binärcode finden. Solche Routinen sind dann als Makros (s. Präprozessorkapitel) implementiert. Neuere Compiler weisen bisweilen die gleiche Funktion wahlweise als Makro oder als Unterprogramm (die Namen sind dann leicht unterschiedlich) auf. Der Grund liegt darin, daß die Makroversionen Probleme mit Seiteneffekten – z.B. mit dem Verändern von Pointerwerten im Aufruf – hervorrufen können. Beliebt sind solche parallelen Versionen vor allem im Bereich der Test- und Konvertierfunktionen wie *isascii(), tolower()* etc. Anständige Compilerhandbücher weisen darauf hin, welche Funktionen als Makros und welche als echte Unterprogramme implementiert sind.

12.2 Die wichtigsten Funktionen der C-Bibliothek

Die nachfolgend beschriebenen Bibliotheksfunktionen sind alphabetisch sortiert, weil ich glaube, daß man so leichter Informationen über eine Funktion finden kann. In vielen Handbüchern wird eine eher logisch orientierte Sortierung vorgenommen, die ich hier der Vollständigkeit halber wiedergebe. Aufgeführt werden nur die Routinen, die weiter unten beschrieben werden. Demnach werden die Bibliotheksfunktionen in die folgenden Gruppen eingeteilt:

- Terminal-IO : gets, getchar, puts, putchar, printf, scanf
- Standard-IO: fopen, fclose, fgets, fputs, fprintf, fgetc, fputc, fscanf
- Basis-IO (UNIX-IO): open, close, read, write, creat, lseek
- Zeichenklassifizierung: isascii, isdigit, islower, isupper
- Zeichenkonvertierung: atoi, atol, tolower, toupper
- Mathematische Routinen: sin, cos, tan, sqrt, abs
- String-Routinen: strcat, strlen, strcmp, strcpy, strncmp, strncpy, strchr, sprintf, sscanf
- Speicherverwaltungsroutinen: malloc, realloc, free
- Systemfunktionen: chdir, exit, umask, chmod

Für die korrekte Verwendung einiger Funktionen werden bestimmte *#include*-Files erwartet. So verlangen die Routinen zur Zeichenklassifikation und zur Zeichenkonvertierung den File *ctype.h*. Die mathematischen Funktionen erwarten *math.h*.

Die Bibliotheksfunktionen werden am Anfang jeder Beschreibung aus **Compilersicht** deklariert. Das heißt nun nicht, daß der Benutzer diese Deklarationen übernehmen muß – im Gegenteil: diese Deklarationen erscheinen so oder ähnlich in stdio.h und sonstigen *#include*-Files. Die Deklaration soll lediglich einen Hinweis darauf geben, welchen Rückgabewert die jeweilige Funktion liefert (in C sind alle Prozeduren Funktionen!) und wie die Argumente 'getypt' sind. Ich habe einige Male den Kunsttyp ANYTYPE verwendet. Er soll darauf hinweisen, daß der genaue Typ von der Verwendung der Funktion abhängt und mitunter erst zur Laufzeit (also erst nach Untersuchung der konkreten Verwendung der Routine) bestimmt werden kann.

```
abs: Absolutwert einer int-Zahl

Deklaration: int abs(zahl)
             int zahl;
```

abs liefert den Absolutwert einer *int*-Zahl. *abs*(-3) liefert z.B. den Wert 3.

```
atoi: Konvertierung eines Strings in eine Int-Zahl

Deklaration: int atoi(str)
             char *str;
```

Der String *str* darf führende Blanks, Tabs sowie ein Vorzeichen enthalten. Ansonsten darf er nur aus numerischen Zeichen (kein Punkt oder Komma!) bestehen. Die Routine liefert das konvertierte Ergebnis.

```
atol: Konvertierung eines Strings in eine long-Zahl

Deklaration: long atol(str)
             char *str;
```

Das Verhalten von *atol* ist (bis auf den Rückgabewert) gleich *atoi*.

```
atof: Konvertierung eines Strings in eine float-Zahl

Deklaration: double atof(str)
             char *str;
```

Das Verhalten der Funktion entspricht sinngemäß *atoi*. Allerdings ist in *str* noch das Dezimalzeichen '.' sowie das Exponentialzeichen 'e' oder 'E' zulässig.

```
chdir: Wechsel des aktuellen Directorys

Deklaration: int chdir(str)
             char *str;
```

str ist der Pfadname des neu anzuwählenden Directories. Die Funktion liefert -1 im Fehlerfall.

```
chmod: Aendern der Zugriffsrechte

Deklaration: int chmod(str,maske)
             char *str;
             unsigned maske;
```

Mit dieser Funktion können die Zugriffsrechte (Schreibrechte, Leserechte, Ausführungsrechte) für Dateien und Verzeichnisse geändert werden. *str* muß ein gültiger Dateiname (Pfadname) sein. *maske* ist die Maske der zu erlaubenden Zugriffsrechte. Gesetzte Bits entsprechen erlaubten Rechten – im Gegensatz zu *umask()* (s. dort). *chmod()* liefert Null, wenn kein Fehler aufgetreten ist und *EOF* im Fehlerfall, wenn z.B. der Pfadname nicht korrekt ist oder wenn nicht genügend Privilegien vorhanden sind, um die Zugriffsrechte umzudefinieren. Beispiel s. Kap. 7.

```
close: Schliessen eines Files

Deklaration: int close(fno)
             int fno;
```

Diese Funktion schließt den File mit der Filenummer fno. Die Funktion gehört zur Serie *open() – read() – write() – creat(),* also zum Basis-IO. Im Fehlerfall (falls *fno* keinem geöffneten File entspricht) wird -1 geliefert, sonst ein Wert >=0. Noch nicht geschriebene Daten werden automatisch weggeschrieben.

```
cos: den Cosinus eines Arguments im Bogenmass ermitteln

Deklaration: double cos(dvar)
             double dvar;
```

cos() ermittelt den Cosinus seines Arguments, das im Bogenmaß angegeben wird. Der File *math.h* sollte einkopiert werden. Das folgende Beispiel zeigt den Aufruf der wichtigsten trigonometrischen Funktionen mit Umrechnung von Grad in Bogenmaß (radians):

```
#include <stdio.h>
#include <math.h>
main()
{
double dv,pi;
  pi = 3.14159;
  dv = sin(45.0 * pi / 180.0);
  printf("\nsin(45)=%g",dv);
  dv = cos(45.0 * pi / 180.0);
  printf("\ncos(45)=%g",dv);
  dv = tan(45.0 * pi / 180.0);
  printf("\ntan(45)=%g",dv);
}
```

```
creat: Anlegen einer neuen Datei

Deklaration: int creat(name,modus)
             char *name;
             int modus;
```

creat() gehört zur Serie der Basis-IO Funktionen. Die Funktion liefert im Fehlerfall 1 und sonst eine Filenummer, die für weitere Operationen verwendet werden kann. Nach Aufruf von *creat()* ist der File für Schreibbetrieb geöffnet. Fall der angesprochene File schon existierte, wird die interne Schreib-/Lesemarke auf den Fileanfang gesetzt. Der Parameter *modus* darf nicht mit dem ähnlichen Parameter der Funktion *open()* verwechselt werden. Er gibt hier an, welche Zugriffsrechte verschiedene Benutzergruppen auf die geschaffene Datei haben (s. Kap. 7). Der Parameter *modus* wird mit der Standardzugriffsmaske des Systems oder mit einer durch *umask()* (s.dort) eingestellten Maske verknüpft. Details müssen dem jeweiligen Compilerhandbuch entnommen werden. Die Funktion *open()* führt in einigen Versionen der C-Bibliothek ein implizites *creat()* durch.

```
exit: Beenden eines Programms mit Statusrückgabe

Deklaration: int exit(rcode)
             int rcode;
```

Mit *exit()* kann an einer beliebigen Stelle eines Programms der Programmlauf beendet werden und dem rufenden Programm (meist der Kommandointerpreter) ein Status *rcode* mitgeteilt werden. In UNIX hat *rcode* den Wert 0 bei korrektem Ablauf und sonst einen Wert ungleich Null. *exit()* bewirkt unterschiedliche Endebehandlungen, z.B. Schließen von Files, Freigeben von allokiertem Speicher etc.

```
fclose: Schließen eines Files

Deklaration: FILE *fclose(fpointer)
             FILE *fpointer;
```

Diese Routine schließt den mit *fpointer* bezeichneten File. Falls die dem File zugeordneten Puffer noch nicht vollständig ausgeschrieben wurden, wird dies jetzt getan. Die Routine liefert *NULL* im Fehlerfall. Die Standardfiles *stdin*, *stdout* und *stderr* brauchen im Normalfall nicht geschlossen zu werden.

```
fgetc: Lesen eines Zeichens aus einem File.

Deklaration: int fgetc(fpointer)
             FILE *fpointer;
```

fgetc() gehört zur Serie des zeichenorientierten IO der Standardroutinen. Die Funktion liest das jeweils nächste Zeichen aus einem File. Sie liefert -1 im Falle von *EOF* oder bei einem Lesefehler. *getc()* ist die Makrovariante von *fgetc()*. *getchar()* ist meist als *fgetc(stdin)* definiert. Man sollte unbedingt beachten, daß der Rückgabewert ein *int*-Wert ist und kein *char*-Wert. Ein beliebter Fehler besteht nämlich darin, den Rückgabewert von *fgetc()* einer *char*-Variablen zuzuordnen und dann auf -1 zu prüfen. Dieser Vergleich geht nie auf, wenn der Compiler nicht per default *signed* Characters erzeugt. Beim Lesen von CTL-Z (oder CTL-C oder CTL-Y etc.) liefern einige Versionen ebenfalls den Wert -1, also *EOF*. *fgetc()* ist also nicht immer transparent.

```
fgets: Lesen einer Zeile aus einem File

Deklaration: char *fgets(ziel,max,fpointer)
             char *ziel;
             int max;
             FILE *fpointer;
```

fgets() liefert eine Zeile aus einem File. Das Zeilenende LF bleibt (im Gegensatz zu *gets()*) in der gelesenen Zeile. *max* ist die maximale Anzahl Zeichen, die in *ziel* abgelegt werden sollen. Die Zeile wird als String abgelegt, d.h. mit einer binären Null beendet. *fgets()* liefert die Adresse *ziel* im Normalfall und im Fehlerfall oder bei EOF den Wert *NULL*.

```
fopen: Oeffnen eines Files

Deklaration: FILE *fopen(fname,mstring)
             char *fname;
             char *mstring;
```

fopen() öffnet den File mit dem Namen *fname* und der in *mstring* angegebenen Betriebsart (Lesen, Schreiben). Die Funktion liefert *NULL* im Fehlerfall und sonst einen Pointer auf eine Struktur vom Typ *FILE*, der bei den weiteren Operationen (*fgets(), fgetc(), fputs(), fputc(), fclose()*) weiterverwendet wird. Typische Werte für *mstring* sind "w" für Schreibbetrieb und "r" für Lesebetrieb. Ein Eröffnen zum Schreiben legt einen bisher nicht vorhandenen File neu an. Falls der so geöffnete File bisher schon existierte, wird er von Anfang an neu beschrieben. Eine *fopen()*-Operation auf die Standard Ein-/Ausgabegeräte (Tastatur, Schirm) ist nicht nötig und wird bisweilen ungnädig aufgenommen.

```
fprintf: Formatiertes Schreiben in einen File

Deklaration: int fprintf(fpointer,fstr,arg1,arg2,...)
             FILE *fpointer;
             char *fstr;
             ANYTYPE arg1,arg2...;
```

Die Funktion liefert die Anzahl der geschriebenen Zeichen als Wert. Bei einem IO-Fehler liefert sie den Wert -1. Sie verhält sich wie *printf()* (s. dort).

```
fputs: Schreiben einer Zeile in einen File

Deklaration: int fputs(str,fpointer)
             char *str;
             FILE *fpointer;
```

fputs() schreibt den String *str* (terminiert mit einer binären Null) in den durch *fpointer* angegebenen File. An *str* wird (im Gegensatz zu *puts()!)* kein Linefeed angehängt. Dies muß also der Programmierer selbst erledigen. Die binäre Null wird nicht mitgeschrieben. Die Funktion liefert -1 bei einem Schreibfehler.

```
free: Freigeben eines reservierten Speicherbereichs

Deklaration: int free(bereich)
             ANYTYPE *bereich;
```

free() ist das Gegenstück zu *malloc()* Es darf nur Speicher freigegeben werden, der über *malloc()* reserviert wurde, da *free()* eine bestimmte (compilerabhängige) interne Datenstruktur des reservierten Blocks auswertet. Die Funktion liefert einen Rückgabewert < 0, wenn ein Fehler aufgetreten ist. Der durch *free()* freigegebene Bereich sollte anschließend durch *malloc()* wieder reserviert werden können, ohne daß es eine Garantie dafür gibt, genau den freigegebenen Bereich zu erhalten.

```
fscanf: Lesen und Konvertieren aus einem File

Deklaration: int fscanf(fpointer,fstr,arg1,arg2,...)
             FILE *fpointer;
             char *fstr;
             ANYTYPE *arg1,*arg2,...;
```

fscanf() liest formatierte Objekte wie Zahlen, Einzelzeichen und Strings aus einem File. Details s. *scanf()*.

```
gets: Lesen einer Zeile vom Schirm

Deklaration: char *gets(buf)
             char *buf;
```

gets() liest eine Zeile, die mit CR (Code 0x0d) beendet ist und legt sie ohne das Endezeichen im Puffer ab, der als Parameter übergeben wird. Statt des CR wird eine binäre Null als Stringende eingetragen. Der Puffer muß ausreichend groß definiert werden. Nicht vergessen, daß die binäre Null auch ein Byte belegt! *gets()* reagiert üblicherweise auf spezielle Steuerzeichen bei der Eingabe, z.B. auf Backspace (0x08), CTL-C (0x03), CTL-Y (0x25), CTL-Z (0x26).

```
getc: Lesen eines Zeichens aus einer Datei

Deklaration: int getc(fpointer)
             FILE *fpointer;
```

getc() ist oft nur eine Makroversion von *fgetc()*. *getchar()* ist meist als *getc(stdin)* deklariert. Details s. *fgetc()*.

```
getchar: Lesen eines Zeichens von der Tastatur

Deklaration: int getchar();
```

Diese Routine liefert den Integerwert eines Zeichens, das von der Tastatur eingelesen wird. Wenn keine besonderen Vorkehrungen (in UNIX Konfiguration über *ioctl()*) getroffen werden, wird das gelesene Zeichen auch auf dem Schirm angezeigt (Echo).

getchar() sollte bei Eingabe eines *EOF*-Zeichens (z.B. CTL-Z) -1 als Rückgabewert liefern. Die Eingabe über *getchar()* ist also nicht notwendigerweise transparent.

```
isascii: Pruefen, ob ein ASCII-Zeichen vorliegt

Deklaration: int isascii(c)
             char c;
```

Die Funktion prüft, ob das übergebene Zeichen im ASCII-Codebereich (also von 0x00 bis 0x7f) liegt. Sie liefert einen Wert ungleich Null, wenn dies der Fall ist und sonst Null. Vorsicht: die deutschen Umlaute auf Amiga, Atari ST oder VAX/VMS gelten nicht als ASCII-Zeichen! *isascii()* wird als Makro implementiert. Für die Verwendung von *isascii(), isdigit(), islower(), isupper()* (es gibt noch mehr is-Funktionen) wird der *#include*-File *ctype.h* benötigt. Man sollte sich diesen File genau anschauen.

Ein kurzer Auszug aus einer Version von *<ctype.h>*. Die Tabelle *__atab* wird normalerweise automatisch dazugelinkt, kann jedoch vom Benutzer frei definiert werden. In diesem Fall muß (im vorliegenden Beispiel) ein initialisierter Array des gleichen Namens bereitgestellt werden und die Konstante *CTYPE* definiert werden. Den vorgefertigten Tabellen sollte man mißtrauen, da die Compiler meist aus Amerika kommen und infolgedessen wenig Bewußtsein um Umlautprobleme zu erwarten ist:

```
/*  Bit patterns for character class DEFINEs */
#define __c 01
#define __p 02
#define __d 04
#define __u 010
#define __l 020
#define __s 040
#define __cs    041
#define __ps    042

#ifndef CTYPE
extern  char    __atab[];
#endif
```

```
/*  Character Class Testing and Conversion DEFINEs: */
#define isascii(ch) ((ch) < 0200)
#define isalpha(ch) (__atab[ch] & (__u | __l))
#define isupper(ch) (__atab[ch] & __u)
#define islower(ch) (__atab[ch] & __l)
#define isdigit(ch) (__atab[ch] & __d)
#define isalnum(ch) (__atab[ch] & (__u | __l | __d))
#define isspace(ch) (__atab[ch] & __s)
#define ispunct(ch) (__atab[ch] & __p)
#define isprint(ch) (__atab[ch] & (__u | __l | __d | __p))
#define iscntrl(ch) (__atab[ch] & __c)

#define tolower(ch) (isupper(ch) ? (ch)+('a'-'A') : (ch))
#define toupper(ch) (islower(ch) ? (ch)+('A'-'a') : (ch))
#define toascii(ch) ((ch) & 0177)
```

```
isdigit: Prueft ob eine ASCII-Ziffer vorliegt

Deklaration: int isdigit(c)
             char c;
```

Die Funktion prüft, ob das übergebene Zeichen im numerischen ASCII-Bereich (0x30 - 0x39) liegt. Sie liefert einen Wert ungleich Null, wenn dies der Fall ist und sonst Null. *isdigit()* wird als Makro implementiert. Weiteres s. *isascii()*.

```
islower: Prueft, ob ein Kleinbuchstabe vorliegt

Deklaration: int islower(c)
             char c;
```

Die Funktion prüft, ob ein Kleinbuchstabe vorliegt. Sie liefert einen Wert ungleich Null, wenn dies der Fall ist. Die Funktion ist als Makro implementiert. Zum Problem der Anpaßbarkeit s. *isascii()*.

```
isupper: Prueft, ob ein Grossbuchstabe vorliegt

Deklaration: int isupper(c)
             char c;
```

Wenn *c* ein Großbuchstabe ist, liefert die Funktion einen Wert ungleich Null. Die Funktion ist als Makro implementiert. Zum Problem der Anpaßbarkeit s. *isascii()*.

```
log: den natürlichen Logarithmus berechnen

Deklaration: double log(dvar)
             double dvar;
```

log() berechnet den natürlichen Logarithmus (Basis: die Zahl *e*) seines Arguments. Der *#include*-File *math.h* sollte angegeben werden.

```
log10: den Zehnerlogarithmus berechnen

Deklaration: double log10(dvar)
             double dvar;
```

log10() liefert den Zehnerlogarithmus seines Arguments. Der #include-File *math.h* sollte einkopiert werden.

```
longjmp: Anspringen eines globalen Sprungziels

Deklaration: int longjmp(env,rcode)
             jmp_buf env;
             int rcode;
```

```
lseek: auf eine beliebige Position innerhalb eines Files posi-
tionieren

Deklaration: long lseek(fno,offset,modus)
             int fno;
             long offset;
             int modus;
```

Diese Funktion gehört in die Serie der Basisprozeduren (UNIX-IO). Sie ermöglicht das wahlfreie Positionieren auf beliebige Positionen eines Files. Für Aufgaben wie ISAM-Dateiverwaltung ist diese Routine also ideal. Man beachte, daß der Offset als *long*-Zahl angegeben werden muß. Der Rückgabewert (ebenfalls *long!*) ist -1 im Fehlerfall oder die erreichte absolute Byteposition innerhalb des angegebenen Files. Für *modus* sind im allgemeinen drei Werte zulässig: 0 für absolutes Positionieren vom Fileanfang aus, 1 für relatives Positionieren von der aktuellen Position aus und 2 für Positionieren vom Fileende aus. Da diese Routine stark an der UNIX-Philosophie vom File als einer einfachen Folge von Bytes orientiert ist, können sich in anderen Betriebssystemen Probleme mit *lseek()* ergeben. Deshalb bitte das jeweilige Compilerhandbuch konsultieren.

```
malloc: Speicherplatz reservieren

Deklaration: char *malloc(anz)
             unsigned int anz;
```

Diese Funktion reserviert die in *anz* angegebene Anzahl von Bytes für den gerade laufenden Prozeß. Sie liefert *NULL* wenn kein Platz mehr da ist oder aber die Anfangsadresse des reservierten Bereichs. Die Funktion garantiert bei wiederholten Aufrufen nicht, daß der reservierte Platz zusammenhängend oder auch nur aufsteigend angelegt wird. Intern wird für die Verwaltung des angeforderten Speichers zusätzlich Platz reserviert, so daß in Wirklichkeit mehr Platz reserviert wird als vom Programmierer angegeben. Der reservierte Platz kann mit der Funktion *free()* (s. oben) wieder freigegeben werden. Wenn die interne Verkettung der durch *malloc()* reservierten Bereiche durch Programmierfehler zerstört wurde, kann es zu katastrophalen Fehlern kommen (die Sie aber nicht C anlasten sollten). Wieviel Platz durch *malloc()* reserviert werden kann, ist systemabhängig (Größe des Heapsegments). Üblicherweise wird der durch *malloc()* reservierte Speicher bei Beendigung des Programms dem System wieder zur Verfügung gestellt, falls dies nicht schon durch *free()* erfolgte.

```
printf: formatierte Ausgabe auf den Schirm

Deklaration: int printf(fstring,arg1,arg2,arg3,...)
             char *fstring;
             ANYTYPE arg1,arg2,arg3,...;
```

printf() ist eine der komplexesten und meist verwendeten Routinen der C-Bibliothek. Sie hat mindestens ein Argument *fstring* (für Formatstring), kann aber mit beliebig vielen ($n >= 1$) Argumenten aufgerufen werden. Die Funktion liefert als Wert die Anzahl der geschriebenen Zeichen. Im Falle eines IO-Fehlers (bei Umlenkung der Ausgabe denkbar) wird -1 geliefert. Der Formatstring *fstring* weist Zeichen auf, die direkt ausgegeben werden und Formatieranweisungen, die mit dem Zeichen % beginnen. Falls das Prozentzeichen % selbst ausgegeben werden soll, muß es als %% notiert werden. Wenn in einem Formatstring n Formatieranweisungen enthalten sind, benötigt *printf()* mindestens $n+1$ Argumente. Der Mechanismus der Abarbeitung ist einfach: der Formatstring wird zeichenweise interpretiert. Wenn ein % gefunden wird und das folgende Zeichen nicht wieder ein % ist, wird das jeweils nächste Argument gemäß der Formatieranweisung ausgegeben. Eine einfache eigene Implementierung von *printf()* ist im Kapitel über die Funktionen in C angegeben. *printf()* erzeugt normalerweise selbst für einfache Programme einen sehr großen Code, da die Zahl der möglichen Formate doch sehr groß ist. Der Formatstring kann übrigens auch in einem Array übergeben werden – er muß also nicht aus einer Stringkonstanten bestehen. Dies erklärt übrigens auch, weshalb die nicht benötigten

Formate nicht einfach 'wegoptimiert' werden können: wenn der Formatierstring in einem Array steht und z.B. von einem File eingelesen wurde, kann der beste Optimierer nicht wissen, welches Format aktuell benutzt werden soll!

Eine Formatieranweisung wird durch das Zeichen % eingeleitet und endet mit einem typähnlichen Kennbuchstaben, z.B. *d, x, u,s.* Zwischen Einleitungszeichen und Endezeichen werden die Details der Formatierung angegeben, z.B. ob das Argument vom Typ *long* ist (z.B. *%ld*), wieviele Stellen auszugeben sind (z.B. *%5d*) , ob vorlaufende Nullen auszugeben sind (*%05d*). Der folgende Überblick der Kennbuchstaben beschränkt sich auf die in K&R angegebenen Typen:

- c: Das Argument wird als Character (Ein-Byte Größe) ausgegeben.
- d: Das Argument wird als vorzeichenbehaftete ganze Dezimalzahl ausgegeben.
- e: Das Argument (*float* oder *double*) wird als Fließkommazahl in wissenschaftlicher Darstellung ausgegeben.
- f: Das Argument (*float* oder *double*) wird als Fließkommazahl in der üblichen Notation ausgegeben.
- g: Das Argument (*float* oder *double*) wird im *e*- oder im *f*-Format ausgegeben, je nach dem, welche Darstellung weniger Druckstellen ergibt.
- o: Das Argument wird als Oktalzahl ausgegeben. Die in C-Konstanten übliche vorlaufende 0 als Kennung für Oktalzahlen wird nicht automatisch erzeugt!
- s: Das Argument (Pointer) wird als String interpretiert.
- u: Das Argument wird als vorzeichenlose (*unsigned*) ganze Dezimalzahl interpretiert.
- x: Das Argument wird als Hexadezimalzahl ausgegegeben. Das in C-Hexkonstanten übliche Präfix *0x* wird nicht automatisch erzeugt!

Beispiel:

```
/* printf - Tests */
#include <stdio.h>
main()
{
int ivar        = -123;
unsigned uvar   = 34567;
long lvar       = 123567L;
char cc         = 'A';
char cstring[10];
float fvar      = 1.414213562;
  printf("\n%c",cc);
  printf("\n%d %06d 0x%x 0%o",ivar,ivar,ivar,ivar);
  printf("\n%u %07u 0x%x 0%o",uvar,uvar,uvar,uvar);
```

```
  printf("\n%ld 0x%08lx 0%010lo",lvar,lvar,lvar);
  printf("\n%f %e %g",fvar,fvar,fvar);
  printf("\n%.7f %.7e %.7g",fvar,fvar,fvar);
  strcpy(cstring,"Hallo!");
  printf("\n%s",cstring);
  printf("\n%10s",cstring);
}   /* end main */
```

```
Das Ergebnis:

A
-123 -00123 0xff85 0177605
34567 0034567 0x8707 0103407
123567 0x0001e2af 00000361257
1.414214 1.414214e+000 1.414214
1.4142135 1.4142135e+000 1.4142135
Hallo!
    Hallo!
```

putc: Ausgabe eines Zeichens in eine Datei

```
Deklaration: int putc(c,fpointer)
             char c;
             FILE *fpointer;
```

putc() liefert EOF im Fehlerfall. *putc()* ist nur die Makroversion von *fputc()* (s. dort).

putchar: Schreiben eines Zeichens auf den Schirm

```
Deklaration: int putchar(c)
             int c;
```

Im Unterschied zu *getchar()* ist die Arbeitsweise von putchar bei allen Compilern ungepuffert. *putchar()* wartet also nicht auf ein CR oder auf das Füllen eines Puffers, bis die Zeichen am Schirm erscheinen. Ein LF als Ausgabezeichen wird meist als CR + LF ausgegeben, d.h. das CR wird zusätzlich erzeugt. Aber auch dies ist installationsabhängig und kann bisweilen konfiguriert werden. *putchar()* interpretiert üblicherweise die ASCII-Kontrollcodes. Auch *putchar()* wird manchmal als Makro implementiert. Im Fehlerfall liefert *putchar()* den Wert -1.

```
puts: Schreiben einer Zeile auf den Schirm

Deklaration: int puts(str)
             char *str;
```

puts() gibt die übergebene Zeile ohne Interpretation auf dem Schirm aus. Die Zeile muß wie üblich mit einer binären Null beendet sein. Da der Ausgabestring von *puts()* nicht groß interpretiert wird, kann *puts()* bei einfachen Anwendungen schneller sein als *printf()*. An den Ausgabestring *str* wird ein Linefeed angehängt (im Gegensatz zu *fputs)*. *puts()* liefert -1 im Fehlerfall.

```
open: Oeffnen einer Datei fuer Block-IO

Deklaration: int open(name,modus)
             char *name;
             int modus;
```

open() gehört zur Gruppe der Basisprozeduren (UNIX-IO). *name* ist der Name des zu öffnenden Files als String. *modus* gibt an, in welcher Betriebsart (0 = Lesen, 1 = Schreiben, 2 = Lesen und Schreiben) betrieben werden soll. *open()* liefert eine Filenummer, die in den weiteren Operationen *(read(), write(), close())* benutzt wird. Diese Filenummer sollte größer als 2 sein, da die Filenummern 0 bis 2 für die Standardeingabe, die Standardausgabe und die Fehlerausgabe vorbelegt sind. Die genannten drei Standardfiles dürfen / können nicht explizit geöffnet werden. Bisweilen gibt es Varianten von *open()* wie *opena()* (Öffnen für reine ASCII-Daten) und *openb()* (Öffnen für Binärdaten).

```
read: Lesen eines Blocks aus einer Datei

Deklaration: int read(fno,puffer,anz)
             int fno;
             char *puffer;
             int anz;
```

read() gehört zur Gruppe der Basisprozeduren (UNIX-IO). Aus dem File mit der Nummer *fno* werden maximal *anz* Bytes gelesen und im Puffer *puffer* abgelegt. *read()* liefert nicht unbedingt *anz* Zeichen, sondern eventuell auch weniger. An das Ende des Puffers wird keine binäre Null gehängt! Rückgabewert ist die Zahl der gelesenen Zeichen. Falls der Rückgabewert 0 ist, wurde *EOF* erreicht. Falls *anz* im Compilerhandbuch als *unsigned int* angegeben ist, ist der Rückgabewert auch *unsigned int*. Mit *read()* kann auch von der Tastatur (*fno = 0*) oder von einem sonstigen peripheren Gerät gelesen werden.

```
realloc: Neuanlegen eines Speicherblocks

Deklaration: char *realloc(oldpoi,groesse)
             char *oldpoi;
             int groesse;
```

realloc() versucht, für einen bereits über *malloc()* angelegten Speicherblock mit der Adresse *oldpoi* einen neuen Platz zu finden, der die Größe in Bytes des Arguments *groesse* hat. *realloc()* liefert *NULL*, falls dies nicht gelingt und sonst einen Pointer auf den neuen Speicherblock, dessen Adresse nicht mit *oldpoi* übereinstimmen muß (aber kann). Der alte Speicherblock *oldpoi* wird freigegeben.

```
scanf: Lesen vom Standardinput und Konvertieren

Deklaration: int scanf(fstr,arg1,arg2,...)
             char *fstr;
             ANYTYPE *arg1,*arg2,...;
```

Wenn transportabel programmiert werden soll, empfiehlt es sich, die Erkennung von Feldgrenzen selbst zu programmieren und *scanf()* oder seine Verwandten nur für einfache Konvertierungen einzusetzen. Die nachfolgenden Erläuterungen beziehen sich daher auf den Idealfall; auf realen Systemen können Abweichungen auftreten. Nur der in K&R definierte Umfang der Eigenschaften von *scanf()* wird hier beschrieben.

scanf() arbeitet gleichsam umgekehrt wie *printf()*. Es weist mindestens ein Argument – den Formatstring *fstr* – auf, kann aber soviele weitere Argumente haben, wie der Formatstring Konvertieranweisungen enthält. Der Formatstring enthält Konstantzeichen und Konvertieranweisungen, die mit dem Zeichen % beginnen. Wenn Konstantzeichen im Formatstring enthalten sind, dann müssen sie mit den jeweils gelesenen Zeichen übereinstimmen, sofern diese nicht einem Weißraum (Blank, Tabulator, Linefeed) entsprechen. Die Inputdaten stellen Sie sich am besten feldweise organisiert vor: jedes Feld wird von Konstantzeichen oder von Weißraum begrenzt. Der Weißraum muß nicht explizit im Formatstring erscheinen. Da ein Zeilenende wie ein Weißraum behandelt wird, liest *scanf()* über Zeilenenden hinaus. Die Funktion liefert als Wert die Anzahl der konvertierten Inputobjekte. Alle Argumente von *scanf()* müssen Adressen sein! In der Praxis werden Sie für Einzelvariablen den Adreßoperator verwenden müssen.

Die Konvertieranweisungen beginnen wie in *printf()* mit % und können die folgenden Kennbuchstaben aufweisen:

- d, o, x: Argument sollte vom Typ *int* * sein.
- ld, lo, lx: Argument vom Typ *long* *.
- c: Argument vom Typ *char* *.
- s: Argument sollte *char*-Array sein (Platz für binäre Null nicht vergessen).
- e, f: Argument vom Typ *float* *.
- le, lf: Argument vom Typ *double* *.

Ein kleines Beispielprogramm, das zum Erweitern und Experimentieren gedacht ist:

```
/* scanf - Tests */

#include <stdio.h>
main()
{
int ivar1,ivar2,ivar3,iret;
  while (1)
  {
    ivar1 = ivar2 = ivar3 = -1;
    iret = scanf("%d,%d;%d",&ivar1,&ivar2,&ivar3);
    printf("\niret=%d ivar1=%d ivar2=%d ivar3=%d\n",
              iret,ivar1,ivar2,ivar3);
    if (iret <= 0) break;
  }
}   /* end main */


Ergebnisse:

10 20 30<CR>
iret=1 ivar1=10 ivar2=-1 ivar3=-1
iret=1 ivar1=20 ivar2=-1 ivar3=-1
iret=1 ivar1=30 ivar2=-1 ivar3=-1
10,20;30<CR>
iret=3 ivar1=10 ivar2=20 ivar3=30
10 ,20 ; 30<CR>
iret=1 ivar1=10 ivar2=-1 ivar3=-1
iret=1 ivar1=-1 ivar2=20 ivar3=-1
iret=0 ivar1=-1 ivar2=-1 ivar3=-1
```

```
sin: den Sinus eines Arguments im Bogenmass ermitteln

Deklaration: double sin(dvar)
             double dvar;
```

sin() ermittelt den Sinus seines Arguments, das im Bogenmaß angegeben wird. Der File *math.h* sollte einkopiert werden. Beispiel s. *cos()*.

```
sprintf: Formatierte Ausgabe in einen String

Deklaration: int sprintf(dest,fstr,arg1,arg2,...)
             char *dest;
             char *fstr;
             ANYTYPE arg1,arg2,...;
```

Die Funktion liefert die Anzahl der in *dest* abgelegten Zeichen. Der in *dest* abgelegte String wird mit einer binären Null abgeschlossen. Die Funktion liefert -1 im Falle eines internen Pufferüberlaufs. Sie verhält sich sonst wie *printf()* (s. dort). Beispiel:

```
char hbuf[10];
  sprintf(hbuf,"%05d",123);
  printf("\n%s",hbuf);

Ergebnis: 00123
```

```
sqrt: liefert die Quadratwurzel einer Zahl

Deklaration: double sqrt(fn)
             double fn;
```

Beim Gebrauch dieser Funktion sollte *math.h* einkopiert werden.

```
sscanf: Lesen aus einem String und Konvertieren

Deklaration: int sscanf(quelle,fstr,arg1,arg2,...)
             char *quelle;
             char *fstr;
             ANYTYPE *arg1,*arg2,...;
```

Die Funktion verhält sich wie *scanf()* (s.dort). Es wird allerdings aus einem String statt vom Terminal gelesen.

```
strchr: Prueft, ob ein Zeichen in einem String enthalten
ist

Deklaration: char *strchr(str,c)
             char *str,c;
```

Die Funktion liefert *NULL* für den Fall, daß c nicht im String *str* enthalten ist. Falls *c* in *str* enthalten ist, liefert sie die Adresse des Vorkommens von *c* in *str*.

```
strcmp: Vergleich zweier Strings

Deklaration: int strcmp(str1,str2)
             char *str1,*str2;
```

Die Funktion prüft, ob String *str1* kleiner, gleich oder größer als String *str2* ist. Sie liefert einen Wert < 0 wenn str1 < str2; sie liefert den Wert 0 wenn *str1* = *str2* und einen Wert > 0 wenn *str1* > *str2* ist. Als Vergleichsoperation wird intern die Subtraktion der jeweils verglichenen ASCII-Codes der Einzelzeichen verwendet. Deshalb Vorsicht bei den deutschen Umlauten!.

Beispiel: strcmp("ABC","abc") liefert den Wert -32

```
strcpy: Kopieren eines Strings

Deklaration: int strcpy(ziel,quelle)
             char *ziel,*quelle;
```

Die Funktion kopiert den String mit der Adresse *quelle* in den String mit der Adresse *ziel*. Für *ziel* darf normalerweise kein konstanter String (in "") eingesetzt werden, wohl aber für *quelle*.

```
strlen: Die Laenge eines Strings berechnen

Deklaration: int strlen(str)
             char *str;
```

Die Funktion liefert die Länge des Strings mit der angegebenen Adresse. Die Länge des Strings ist die Anzahl der Zeichen im String ohne die abschließende binäre Null.

```
strncpy: Eine Anzahl Zeichen eines Strings kopieren

Deklaration: int strncpy(ziel,quelle,anz)
             char *ziel,*quelle;
             int anz;
```

Die Funktion liefert die gleichen Rückgabewerte wie *strcpy()*. Wenn in einem der beiden Strings ein Stringendezeichen (binäre Null) angetroffen wird, stoppt die Kopieroperation.

```
strncmp: Eine Anzahl Zeichen zweier Strings vergleichen

Deklaration: int strncmp(str1,str2,anz)
             char *str1,*str2;
             int anz;
```

Die Funktion liefert die gleichen Rückgabewerte wie *strcmp()*. Wenn in einem der beiden Strings ein Stringendezeichen (binäre Null) angetroffen wird, stoppt die Vergleichsoperation.

```
tan: den Tangens eines Arguments im Bogenmass ermitteln

Deklaration: double tan(dvar)
             double dvar;
```

tan() ermittelt den Tangens seines Arguments, das im Bogenmaß angegeben wird. Der *#include*-File *math.h* sollte einkopiert werden. Beispiel s. *cos()*.

```
tolower: Konvertiert einen Grossbuchstaben in einen
         Kleinbuchstaben

Deklaration: char tolower(c)
             char c;
```

Diese Funktion konvertiert einen ASCII-Großbuchstaben in einen ASCII-Kleinbuchstaben. Die deutschen Umlaute können wegen der Beschränkung auf ASCII Probleme bereiten. Wenn die Funktion als Makro implementiert ist, sollte man Seiteneffekte vermeiden.

```
toupper: Konvertiert einen Kleinbuchstaben in einen
         Grossbuchstaben

Deklaration: char toupper(c)
             char c;
```

Diese Funktion konvertiert einen ASCII-Kleinbuchstaben in einen ASCII-Großbuchstaben. Die deutschen Umlaute können wegen der Beschränkung auf ASCII Probleme bereiten. Wenn die Funktion als Makro implementiert ist, sollte man Seiteneffekte vermeiden.

```
umask: Definition einer File-Zugriffsmaske

Deklaration: int umask(maske)
             unsigned int maske;
```

umask() dient dazu, für den laufenden Prozeß (das laufende Programm) eine Maske festzulegen, die beim Anlegen neuer Dateien die Zugriffsrechte festlegt. Die Funktionen *creat(), open()* und *fopen()* verwenden die durch *umask()* festgelegte Maske. In der Maske entsprechen gesetzte Bits verbotenen Rechten! Beispiel: s. Kap. 7

```
write: Schreiben in eine Datei

Deklaration: int write(fno,puffer,anz)
             int fno;
             char *puffer;
             int anz;
```

write() gehört zur Gruppe der Basisprozeduren (UNIX IO). Aus dem Puffer *puffer* sollen *anz* Bytes in den File mit der Filenummer *fno* geschrieben werden. Der Typ der Pufferlänge *anz* ist bisweilen auch *unsigned int.* Es deutet generell auf einen Fehler hin, wenn der Rückgabewert von *write()* nicht gleich *anz* ist. Die Ausführung von *write()* garantiert nicht immer, daß die Daten tatsächlich auch geschrieben wurden. Mit *write()* kann man auch auf die Standardausgabe oder auf ein peripheres Gerät schreiben.

12.3 Ein Testprogramm für Bibliotheksfunktionen

Das nachstehende Testprogramm für Bibliotheksfunktionen hat zwei Aufgaben: einmal soll es dem Leser die Gelegenheit geben, die wichtigsten Funktionen in einem konkreten (wenn auch simplen) Programm verwendet zu sehen. Aus meiner Erfahrung hilft das mehr als noch so kluge Beschreibungen. Eine weitere Funktion liegt darin, einen Rahmen für das Testen der eigenen C-Bibliothek bereitzustellen.

Besonders beachten sollten Sie die folgenden Punkte:

- welche Prozeduren der Bibliothek fehlen?
- gibt es Probleme mit den *#include*-Files?
- ist 'ü' ein alphabetisches Zeichen?
- existiert *strchr()* und arbeitet es korrekt?
- arbeiten *read()* und *write()* transparent (wird LF zu LF+CR expandiert?) ?
- wieviele Speicherblöcke kann *malloc()* anlegen?

Das Programm testet nicht alle der aufgeführten Bibliotheksroutinen, kann aber leicht erweitert werden.

```
/****************************************

  Testprogramm fuer Bibliotheksroutinen
           last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Testet die wichtigsten Bibliotheksroutinen

****************************************/

#include <stdio.h>
#include <ctype.h>
#include <math.h>

int n,fno,iret,fd1;
long lret;
double fret,dv,pi;
char puffer[512],*cadr;
char car[10][10];
typedef struct test
{
  char dummy[1000];
  struct test *next;
} TSTRUCT;
TSTRUCT *badr,*start,*last;
```

```
errexit(str)
char *str;
{
  printf("\n\t\t--- %s ---",str);
  exit(1);
}   /* end errexit */

weiter()
{
  printf("\nWeiter:");
  getchar();
}   /* end weiter */

main()
{
  /* Groesse der Datentypen */
  printf("\nsizeof char:     %d",sizeof(char));
  printf("\nsizeof short:    %d",sizeof(short));
  printf("\nsizeof int:      %d",sizeof(int));
  printf("\nsizeof long:     %d",sizeof(long));
  printf("\nsizeof float:    %d",sizeof(float));
  printf("\nsizeof double:   %d",sizeof(double));
  printf("\nsizeof (char *): %d",sizeof(char *));

  /* Konvertierungen */
  weiter();
  iret = atoi("   -123   ");
  printf("\nErgebnis atoi: %d",iret);
  lret = atol("   +123123   ");
  printf("\nErgebnis atol: %ld",lret);
  fret = atof("   -123.30e7   ");
  printf("\nErgebnis atof: %f",fret);
  iret = isascii('A');
  printf("\nErgebnis isascii fuer A: %d",iret);
  iret = isalpha(' ');
  printf("\nErgebnis isalpha fuer : %d",iret);

  /* float und double Test in printf() */
  printf("\n4.5 als float %f",4.5);
  printf("\n4.5 scientific float %e",4.5);
  printf("\n4.5 in g Format %g",4.5);

  /* mathematische Routinen */
  weiter();
  pi = 3.14159;
  dv = sin(45.0 * pi / 180.0);
  printf("\nsin(45)=%g",dv);
  dv = cos(45.0 * pi / 180.0);
  printf("\ncos(45)=%g",dv);
  dv = tan(45.0 * pi / 180.0);
  printf("\ntan(45)=%g",dv);
  dv = log(10.0);
  printf("\nlog(10.0)=%g",dv);
  dv = log10(10.0);
  printf("\nlog10(10.0)=%g",dv);
  dv = sqrt(2.0);
  printf("\nsqrt(2.0)=%g",dv);
```

```
/* Test Basis-IO */
/* mit Test, ob write per default tranparent ist */
weiter();
printf("\nTest Basis-IO");
/* File anlegen */
fd1 = creat("xx.tst",0xffff);
/* 4 Zeichen hineinschreiben */
iret = write(fd1,"abc\n",4);
/* File schliessen */
close(fd1);
/* und wieder oeffnen */
fd1 = open("xx.tst",0);
printf("\nErgebnsi open= %d",fd1);
/* Das geschriebene wieder lesen */
iret = read(fd1,puffer,4);
/* Ausgabe auf Terminal mit Basis-IO */
write(1,"\nErgebnis read:",15);
write(1,puffer,4);
/* pruefen, ob \n zu LF+CR expandiert wurde */
iret = read(fd1,puffer,1);
if (iret <= 0)
  printf("\nwrite-read arbeiten transparent");
else  printf("\nwrite-read arbeiten nicht transparent");
close(fd1);

/* Stringroutinen */
weiter();
printf("\nstrcmp: %d",strcmp("ABC","abc"));
printf("\nstrncmp: %d",strncmp("ABC","abcdef",4));
printf("\nstrcpy: %s",strcpy(puffer,"vwxyz"));
printf("\nstrncpy: %s",strncpy(puffer,"abcdefg",3));
cadr = strchr("ABCDEFG",'D');
printf("\nstrchr: %lx",cadr);
cadr = strchr("ABCDEFG",'X');
printf("\nstrchr: %lx",cadr);

/* Test malloc und free */
weiter();
start = last = NULL;
iret = 0;
/* Bloecke anlegen bis es nicht mehr geht */
while (1)
  {
   badr = malloc(sizeof(TSTRUCT));
   if (badr == NULL) break;
   iret++;
   if (last != NULL) last->next = badr;
   else              start = badr;
   badr->next = NULL;
   last = badr;
  }
```

```
  /* wieder freigeben */
  badr = start;
  while (badr != NULL)
    {
     last = badr;
     badr = badr->next;
     free(last);
    }
  printf("\n%d Bloecke a 1000 Bytes konnten angelegt werden", iret);
}   /* end main */
```

13 Programmierstil in C-Programmen

13.1 Warum Programmierregeln?

C-Programmierer haben leider oft die unschöne Gewohnheit, sehr dicht und komprimiert zu programmieren. Dadurch entstehen Programme, die für Außenstehende kaum lesbar sind. Es ist deshalb kein Wunder, daß C im Rufe steht, nicht besonders gut lesbar zu sein. Die Erfinder Kernighan & Ritchie hatten offenbar eine Vorliebe für kurze und kürzeste Schreibweisen: siehe die Operatoren +=, -= etc. Diese Vorliebe war ganz eindeutig aus der damaligen Programmierpraxis zu verstehen. Als Eingabegeräte waren Lochstreifenstanzer, Kartenstanzer und Bedienkonsolen (Teletypes) vorwiegend vertreten. Wer diese Geräte kennt, weiß es zu schätzen, wenn die Eingaben kurz ausfallen.

In der Zwischenzeit hat sich freilich die Szene geändert. Die Eingabe geschieht unter Bildschirmkontrolle mit Screen-Editoren. Aus dieser Sicht also gibt es keine Notwendigkeit mehr, knappe und platzsparende Programme zu schreiben.

Da die Informatik einen gewaltigen Boom erlebt hat, ist es nötiger denn je geworden, Programme zu schreiben, die weitgehend aus sich selbst heraus verständlich sind. Es wird immer wahrscheinlicher, daß die von Ihnen geschriebenen Programme irgendwann von einem fremden Programmierer gelesen werden müssen. Für ausgesprochene Übungsprogramme gilt das natürlich nicht. Dennoch sollten Sie sich auch in den einfachsten Programmen bemühen, beim Schreiben gewisse Grundregeln einzuhalten.

Die nachstehenden Regeln haben sich in meiner Programmierpraxis als sinnvoll herausgestellt. Das heißt nicht, daß diese Regeln allgemein akzeptiert würden. Was jetzt folgt, ist nicht naturwissenschaftlich begründbar. Es ist vielmehr meine Ideologie des Programmierens in C. Diese Ideologie hat sich in der Praxis des Arbeitens mit C gefestigt. Da es in C ganz einfach ist, katastrophale Programmfehler einzubauen, ist

eine solche Ideologie unentbehrlich. Ohne die nötige Programmierdisziplin werden Sie mit C bittere Überraschungen erleben. Das sollte jedoch nicht als Bemerkung gegen C verstanden werden. Die Lesbarkeit eines Programms halte ich – gerade im Falle von C – für mindestens genau so wichtig wie seine Korrektheit.

Ein wichtiger Hinweis: Meine eigenen Programmierregeln stimmen in einigen Punkten nicht mit den Schreibkonventionen von Kernighan & Ritchie überein. Am Ende des Kapitels wird ein Programm abgedruckt, das den Konventionen von K&R folgt. Sie sollten entscheiden, welche Schreibkonventionen zur besseren Lesbarkeit führen.

13.2 Elementare Strukturregeln

Ein C-Programm sollte grob aus vier Teilen bestehen:

- **Programmkopf**: er besteht aus C-Kommentaren und beschreibt die Funktion des Programmes, gibt den Autor an, erklärt die zugrundeliegenden Algorithmen, gibt Bedienungshinweise, gibt die Produktionsmaschine und den Testcompiler an.
- **Präprozessorteil**: dort erscheinen zuerst die benötigten *#include*-Files, dann die definierten Konstanten und die Makros.
- **Deklarationsteil**: globale (externe) Daten und Funktionen
- **Programmkörper**: die Folge von Befehlen, die ausgeführt werden sollen.

Sie werden in meinen Beispielsprogrammen sehen, daß diese Grobstruktur auch in den Unterprogrammen weitgehend erhalten bleibt. Auch jede Prozedur sollte zuerst einen Kommentarteil aufweisen, der Funktion und Parameter erklärt, dann die Deklaration der Parameter und der lokalen Variablen, dann den Prozedurkörper. Ich versuche, so weit wie möglich Präprozessorkommandos nur am Anfang des Gesamtprogramms zu verwenden. Natürlich lassen sich diese Regeln nicht immer strikt einhalten.

13.3 Zerlegung in Prozeduren

Es hat sich in der Praxis als vorteilhaft erwiesen, ein C-Programm schon beim Design in möglichst kurze und übersichtliche Prozeduren zu zerlegen. Im aufgeführten Beispielprogramm gibt es nebem dem gemeinsamen Rahmenprogramm *main()* die Prozeduren *filedia()* und *rdfilter()*. Als grobe Regel kann gelten: keine Prozedur sollte länger als eine Schreibseite sein. Jede Prozedur hat natürlich einen eigenen Kommentarvorspann, der ihre Funktionsweise erläutert. Da durch die Zerlegung in einzelne Prozeduren der Programmablauf verlangsamt werden kann (ein Aufruf eines Unterprogramms kostet je nach Maschine viel Zeit), sollte schon beim Design, spätestens aber in einer eigenen Optimierungsphase die Einteilung in Prozeduren genau überlegt werden. Programme, die – wie oben – im wesentlichen aus einer großen Schleife

bestehen, sollten nicht die Hauptarbeit der inneren Schleife einer Prozedur übertragen. Aber dies kann nicht als strikte Regel definiert werden, sondern ist vom Einzelfall abhängig.

Im obigen Beispiel hat die Einteilung in Prozeduren nur den Sinn, das Programm überschaubarer zu machen. Die Grobstruktur eines Programms sollte schon beim Lesen von *main()* erkennbar sein. *main()* darf deshalb nicht allzuviel Programmlogik enthalten; es ist für die Lesbarkeit besser, wenn in *main()* nur die Hauptverarbeitungsschritte in Form von Prozeduraufrufen angestoßen werden.

13.4 Wahl von globalen und lokalen Variablen

Der Unterschied zwischen globalen (externen) und lokalen (internen) Variablen ist im wesentlichen der: globale Variablen sind allen Modulen eines Programms bekannt, lokale Variablen gelten nur im einzelnen Modul oder gar nur im aktuellen Block. Gleichnamige lokale Variablen können in unterschiedlichen Modulen oder Blöcken auftreten.

In der Schulinformatik gilt die Regel: möglichst wenig globale Variablen verwenden. Da eventuell viele Module auf solche Variablen zugreifen und sie verändern, verliert der Programmierer leicht die Übersicht über ihren Gebrauch. Die Alternative dazu wird in der Parametrisierung von Modulen gesehen: alle Daten, die vom Modul bearbeitet werden, sollten also als Parameter an das Modul übergeben werden. Dies hat den schönen Nebeneffekt, daß die einzelnen Module universeller werden. Sie sind ja nicht mehr an feste Namen von Variablen gebunden. Die Namen der Formalparameter gelten lokal, sind also vom Aufruf eines Moduls (Aktualparameter) unabhängig.

Wie alle Schulweisheiten ist dies prinzipiell richtig – aber auch nur prinzipiell! Zum einen kann man in C – und dies aus gutem Grund – auf so ziemlich alles über Pointer zugreifen. Die Theorie von der Abgeschlossenheit der Module ist also sehr löchrig. Zum zweiten ist der Zugriff auf übergebene Parameter und auf lokale Variablen (beide werden auf dem Stack angelegt) auf einigen von mir getesteten Maschinen (8 Bit CPUs) sehr viel langsamer als der Zugriff auf globale Variablen. Und zum dritten gibt es Fälle, in denen die Wahlmöglichkeit zwischen globalen (externen) und lokalen Daten gar nicht gegeben ist. Ein Beispiel: Wenn ich lokal einen Datenpuffer von mehreren Kilobytes Größe brauche (z.B. für Matrizenrechnungen) und diesen intern in einem Unterprogramm definiere, so steigt auf einigen Maschinen das Programm (hoffentlich schon) beim Linken aus. Der Grund? Das Stacksegment wird standardmäßig sehr knapp angelegt. Schlimmer sind die Fälle, in denen das Programm dann erst beim Lauf mit 'Stack overflow' aussteigt.

Die folgenden Regeln haben sich in der Praxis als handhabbar gezeigt: Große Datenbereiche – selbst wenn sie nur lokal in einem Modul verwendet werden – werden besser global deklariert. Schleifenvariablen und 'technische' Variablen wie Variablen

für Zwischenergebnisse etc. werden besser lokal definiert. Dies gilt nicht für den Fall, daß zeitkritische Probleme im Unterprogramm behandelt werden! Wenn immer möglich, sollten externe Daten als Parameter an ein Modul übergeben werden. Falls solche Module jedoch sehr häufig und unter zeitkritischen Umständen aufgerufen werden, bleiben sie besser parameterlos.

Wann immer möglich, sollten Veränderungen an globalen Daten in Prozeduren unterbleiben. Es bietet sich an, Module als Funktionen aufzurufen (Funktionen sind Prozeduren, die einen Wert liefern), die dann an der Aufrufstelle eine globale Variable verändern. Bitte beachten Sie, daß im K&R Standard-Funktionen nur elementare Werte (*int,long, double, char*, Pointer) liefern können. Ganze Strukturen oder Arrays können nicht als Wert einer Funktion geliefert werden. Neuere C-Compiler ermöglichen dies. Wenn Sie transportable Programme schreiben wollen, lassen Sie besser die Finger davon! Falls ein Modul dennoch globale Daten verändert, sollten Sie das an der Aufrufstelle und im Modul selbst deutlich vermerken. Natürlich gilt dies auch für den Fall, daß Sie Pointer von Variablen als Argument übergeben und dann im Unterprogramm über den Sternoperator diese Variable verändern.

13.5 Namensvergabe

Viele Programmierer tendieren dazu, in C kurze Namen für Variablen und Programme zu wählen. Das kommt sicher daher, daß C-Programmierer oft noch aus der Zeit 'stammen', als nicht beliebig viel Platz auf Platte und im Speicher zur Verfügung stand. Möglicherweise aber ist der Grund für diese Tendenz auch darin zu suchen, daß in K&R Variablennamen und Programmnamen nur in den ersten acht Stellen signifikant sind. Die modernen C-Compiler weisen sehr viel mehr signifikante Stellen in Namen auf.

Dennoch lassen sich auch mit nur acht signifikanten Namensstellen 'sprechende' (selbsterklärende) Namen für Variablen und Programme bilden. Als Regeln für die Vergabe von Namen haben sich in der Praxis herausgebildet:

- Variablennamen und Programmnamen sollten auf ihren Verwendungszweck hinweisen. Die Wahl aussagekräftiger Namen für die formalen Parameter einer Prozedur kann z.B. Kommentare ersparen, die die Funktion der Parameter erläutern sollen. Wenn eine Variable als Zähler für irgendwas benutzt wird, heißt sie besser *zaehler* als *n*.
- Kurze und wenig aussagekräftige Namen sollten nur für Laufvariablen und 'kurzlebige' Hilfsvariablen gewählt werden.
- Inhaltlich verwandte Variablen und Programmnamen sollten ein gemeinsames Namenspräfix aufweisen, das die Verwandtschaft andeutet. Beispiel: Wenn ein komplexeres Programm mehrere Eingaberoutinen verwendet, könnten alle Ein-

gaberoutinen mit dem Präfix *inp_* beginnen, z.B. *inp_deczahl(), inp_string(), inp_hexzahl().*

- Üblicherweise werden in komplexen Programmen inhaltlich verwandte Routinen zu Modulpaketen (Bibliotheken) zusammengefaßt. Auch in solchen Fällen ist es sinnvoll, die im einem Paket zusammengefaßten Routinen mit einem gemeinsamen Namenspräfix zu versehen. Alle Routinen eines Graphikpakets könnten z.B. das Namenspräfix *gr_* haben.

13.6 Weitere Schreibregeln

Die jetzt folgenden Schreibregeln lassen vielleicht den Verdacht aufkommen, ich sei ein Pedant. Aber bitte glauben Sie mir: wenn Sie gezwungen sind, öfters fremde oder etwas ältere eigene Programme zu lesen, werden Sie auf ähnliche Regeln verfallen.

Ich halte mich jetzt bewußt nicht an die Schreibkonventionen von K&R, obwohl viele professionelle Programme sich in der Schreibweise an K&R orientieren. Warum ich dies nicht tue, soll an dem folgenden kurzen Programm demonstriert werden. Der Inhalt des Beispielprogramms soll hier nicht interessieren. Zunächst die Schreibweise von Kernigan & Ritchie:

```
/* Schreibweise nach Kernighan und Ritchie */
while (diad->len != EOWB) {
  lxadr1 = lxadr2 = lxadr3 = diad->txt;
  lv1 = lv2 = lv3 = diad->len ^ WLAEN;
  if ((lv1 - comp + gast) == lan) {
    swadr = swa;
    if (gast > 0) {
      lv2 = gast;
      alfe = lastwo;
      while (lv2-- > 0) {
        cmpw = (int)*alfe++ - (int)*swadr++;
        if (cmpw > 0) return(NULL);
        if (cmpw < 0) return(NULL);
      }
      lv2 = lv3 - 1;
      lxadr1++;
    }
    while (lv2-- > 0) {
      cmpw = (int)*lxadr1++ - (int)*swadr++;
      if (cmpw > 0) break;
      if (cmpw < 0) break;
    }
```

```
    return(diad);
  }
  diad = lxadr3 + lv3;
  if (diad->txt[0] & WKOMP) {
    if (diad->len != EOWB) {
      comp = 1;
      gast = lv2 = diad->txt[0] ^ WKOMP;
      alfe = lastwo;
      if (lgast > 0) {
        alfe += lgast;
        lv2  -= lgast;
        lxadr2++;
      }
      while (lv2-- > 0) {
        *alfe++ = *lxadr2++;
      }
    }
  }
  else {
    gast = 0;
    comp = 0;
  }
  lgast = gast;
}
```

Ich halte dies für nicht gut lesbar und schlage deshalb die folgende Notation für das gleiche Programm vor:

```
/* verbesserte Schreibweise */
while (diad->len != EOWB)
{
  lxadr1 = lxadr2 = lxadr3 = diad->txt;
  lv1 = lv2 = lv3 = diad->len ^ WLAEN;
  if ((lv1 - comp + gast) == lan)
  {
    swadr = swa;
    if (gast > 0)
    {
      lv2 = gast;
      alfe = lastwo;
      while (lv2-- > 0)
      {
        cmpw = (int)*alfe++ - (int)*swadr++;
        if (cmpw > 0) return(NULL);
```

```
        if (cmpw < 0) return(NULL);
      }
      lv2 = lv3 - 1;
      lxadr1++;
    }
    while (lv2-- > 0)
    {
      cmpw = (int)*lxadr1++ - (int)*swadr++;
      if (cmpw > 0) break;
      if (cmpw < 0) break;
    }
    return(diad);
  }
  diad = lxadr3 + lv3;
  if (diad->txt[0] & WKOMP)
  {
    if (diad->len != EOWB)
    {
      comp = 1;
      gast = lv2 = diad->txt[0] ^ WKOMP;
      alfe = lastwo;
      if (lgast > 0)
      {
        alfe += lgast;
        lv2  -= lgast;
        lxadr2++;
      }
      while (lv2-- > 0)
      {
        *alfe++ = *lxadr2++;
      }
    }
  }
  else
  {
    gast = 0;
    comp = 0;
  }
  lgast = gast;
}
```

Die wichtigste Änderung gegenüber K&R betrifft die Stellung der **Blockklammern** { und }. Ich halte es für lesbarer, wenn öffnende und schließende Klammer in der glei-

chen Spalte erscheinen. Bei K&R wird die öffnende Klammer am Ende des Befehls gesetzt, der den Block einleitet. Die von mir vorgeschlagene Schreibweise entspricht der üblichen Notation von PASCAL-Programmen und anderen blockorientierten Sprachen. Sie soll die Tatsache, daß die Blockklammern { und } ziemlich 'dünn' sind, durch eine identische Spalteneinteilung kompensieren. Bisweilen sieht man Programme, die die Blockklammern über Präprozessorkommandos durch *begin* und *end* ersetzen. Ich glaube, daß das des Guten zuviel ist.

Die schließenden Blockklammern von Programmen, von Schleifen und von *if*-Befehlen sollten mit Kommentaren versehen werden, die den Block eindeutig markieren. Dies gilt besonders für ineinandergeschachtelte Schleifen und *if*-Befehle. An dem obigen Beispielprogramm (verbesserte Version) soll dies wieder gezeigt werden. Bitte schauen Sie sich nacheinander die obige Version und die nachstehende Version an. Ich glaube, Sie stimmen mir zu, daß die Blockendekommentare wesentlich zur Lesbarkeit beitragen, wenn das Programm eine bestimmte Länge übersteigt. Dies gilt erst recht für den Fall, daß ein Programm länger als eine Seite wird:

```
/* Kommentare an den Blockenden */
while (diad->len != EOWB)
{
  lxadr1 = lxadr2 = lxadr3 = diad->txt;
  lv1 = lv2 = lv3 = diad->len ^ WLAEN;
  if ((lv1 - comp + gast) == lan)
  {
    swadr = swa;
    if (gast > 0)
    {
      lv2  = gast;
      alfe = lastwo;
      while (lv2-- > 0)
      {
        cmpw = (int)*alfe++ - (int)*swadr++;
        if (cmpw > 0) goto kwbsu6;
        if (cmpw < 0) goto kwbsu4;
      }   /* while lv2-- > 0 */
      lv2 = lv3 - 1;
      lxadr1++;
    }   /* if gast > 0 */
    while (lv2-- > 0)
    {
      cmpw = (int)*lxadr1++ - (int)*swadr++;
      if (cmpw > 0) break;
      if (cmpw < 0) break;
    }   /* while lv2-- > 0 */
```

```
    return(diad);
  }   /* if ((lv1 -comp + gast) == lan) */
  diad = lxadr3 + lv3;
  if (diad->txt[0] & WKOMP)
  {
    if (diad->len != EOWB)
    {
      comp = 1;
      gast = lv2 = diad->txt[0] ^ WKOMP;
      alfe = lastwo;
      if (lgast > 0)
      {
        alfe += lgast;
        lv2  -= lgast;
        lxadr2++;
      }
      while (lv2-- > 0)
      {
        *alfe++ = *lxadr2++;
      }   /* while lv2-- > 0) */
    }   /* if diad->len != EOWB */
  }   /* if diad->txt[0] & WKOMP */
  else
  {
    gast = 0;
    comp = 0;
  }   /* else */
  lgast = gast;
}   /* while diad->len != EOWB */
```

Bei komplexen Formeln plädiere ich für eindeutige Klammerung durch runde Gruppierungsklammern. Der Verzicht auf unnötige Klammern zeigt zwar, daß der Programmierer die (nicht einfachen) Prioritätsregeln von C beherrscht. Der Leser des Programms steht jedoch wahrscheinlich nicht auf der gleichen Fertigkeitsstufe wie der Programmierer, so daß eine sinnstiftende Klammerung ihn gar nicht erst ins Grübeln über die Reihenfolge der Auswertung einer Formel kommen lässt. Sie sollten es sich ganz einfach angewöhnen, nicht für C-Gurus zu schreiben, sondern für Durchschnittsprogrammierer. Beispiel:

```
a = b >> 3 / 2 + c << 4 * 5;
/* besser */
a = ((b >> 3) / 2) + ((c << 4) * 5);
```

Zwischen Operatoren und ihren Argumenten sollten Blanks eingefügt werden. Das trägt sicherlich zur besseren Lesbarkeit bei. Es hat aber auch den Vorteil, einige Unschönheiten der C-Syntax zu kaschieren. Bekanntlich gibt es in C die Inkrementoperatoren ++ und --. In arithmetischen Ausdrücken wie *(var1*++ + --*var2)* ist die Einfügung eines Blanks vor und hinter dem Plusoperator sicherlich unumgänglich. Blanks tragen aber generell zur besseren Lesbarkeit bei, weil sie die Funktion der kurznamigen Operatoren herausstreichen.

14 Systemprogrammierung

14.1 Begriffsbestimmung

Das folgende Kapitel soll einige Beispiele für Systemprogrammierung in C geben. Ich beanspruche nicht, daß damit die Informatikdisziplin Systemprogrammierung auch nur annähernd vollständig behandelt wird.

Unter 'Systemprogrammierung' wird zweierlei verstanden:

- Erstellung, Pflege und Tuning von Betriebssystemen
- Betriebssystemnahe Programmierung

Ich zeige hier nur Beispiele für den letzten Aspekt. Sie werden vielleicht fragen: Warum ein solches Kapitel in einem Anfängerbuch? Ganz einfach: Ich gehe davon aus, daß jemand, der C lernt, über den Rahmen dessen hinaus gehen will, was in anderen Programmiersprachen üblich ist; sonst könnte er ja auch Pascal oder BASIC lernen. C steht zumindest teilweise zu Recht im Ruf, gut für die Systemprogrammierung geeignet zu sein. Es wäre also schade, wenn dieser Aspekt nicht behandelt würde, der doch zum guten Ruf von C beiträgt.

Systemnahe Programmierung heißt aber: nahe an das jeweilige Betriebssystem herangehen. Ich kann deshalb nicht allgemeingültige Beispielsprogramme präsentieren. Systemnahe Programmierung ist nahezu automatisch **nicht portabel**. Wenn Sie also zuwenig Informationen über das Betriebssystem haben, in dem die vorgestellten Beispiele ablaufen sollen, dann betrachten Sie dieses Kapitel als Reserve: irgendwann werden zumindest einem professionellen Programmierer solche Probleme begegnen.

Je nach Compiler und Zielmaschine werden dem C-Programmierer unterschiedliche Hilfsmittel für die Systemprogrammierung zur Verfügung gestellt. Der MSC-Compiler für den IBM-PC stellt besonders viele Prozeduren und *#include*-Files zur

Verfügung, was auch beim Amiga auffällt. Der DRC-Compiler (Version 1.1) stellt für die gleiche Zielmaschine kaum Hilfsmittel zur Verfügung, wird aber dafür zusammen mit einem Assembler ausgeliefert. Für den ATARI ST sind kaum spezielle Prozeduren erforderlich, es sei denn, Sie wollen Desktop-Anwendungen erstellen.

14.2 Pflege von Systemtabellen

Über Systemtabellen in pflegbarer Form verfügen meist nur Großsysteme – und hier besonders UNIX. C verfügt über alle Sprachmittel, um solche Systemtabellen zu lesen oder zu verändern. Die Tabellen werden in *#include*-Files als Strukturen definiert und können dann als normale C-Objekte manipuliert werden. Einige Tabellen können nur mit besonderen Zugriffsprivilegien verändert werden – aber das hat nichts mit C zu tun.

UNIX verfügt über konfigurierbare Treiber. Der Treiber für die seriellen Schnittstellen (insbesondere für die eigene Terminalleitung) kann über spezielle Tabellen konfiguriert werden. Dazu dient die Prozedur *ioctl(),* die leider nur in UNIX so einfach wie hier gezeigt verwendet werden kann. Ich kann mit diesem Aufruf die Baudrate einstellen, das Protokoll verändern, Zeichenfilterung im Input und Output veranlassen, Betriebssystemeingriffe provozieren (z.B. wenn ich CTRL-C als *break*-Zeichen definiere). Eine besonders wichtige Einstellung ist die Umstellung von zeilenweiser Reaktion auf zeichenweises Reagieren. Die Prozedur *getchar()* liefert normalerweise erst ein Ergebnis, wenn *CR* eingegeben wurde. In einigen Anwendungen wie in Screeneditoren oder in Kommunikationsanwendungen soll das eigene Programm aber sofort auf jeden Tastenanschlag reagieren. Um dies zu erreichen, kann ich die Leitungscharakteristik wie folgt ändern:

```
/* Aenderung der Terminalcharakteristik in UNIX */
#include<termio.h>
int linenr;
    linenr = fileno(stdin);
    /* Struktur lineIO lesen */
    ioctl(linenr,TCGETA,&lineIO);
    /* CTL-C, CTL-Z, Echo ausschalten */
    lineIO.c_lflag = ~ (ICANON | ECHO);
    /* Mindestanzahl Zeichen = 0 */
    lineIO.c_cc[VMIN] = 0;
    /* Timeout = 1ms einstellen */
    lineIO.c_cc[VTIME] = 1;
    /* Struktur zurueckschreiben */
    ioctl(linenr,TCSETA,&lineIO);
```

Mit dem gleichen Aufruf von *getchar()* kann ich nun den Status überprüfen (liegt ein Zeichen an?) und ein Zeichen lesen. *getchar()* wird eine Null liefern, wenn kein Zei-

chen anliegt und sonst einen Wert ungleich Null. Zudem ist *getchar()* jetzt transparent – CTRL-Z liefert nicht mehr den Wert *EOF*. Wenn ich das Gleiche in MS-DOS oder TOS erreichen will, muß ich verschiedene Betriebssystemroutinen statt eines einzigen Aufrufs von *getchar()* verwenden. Sie sehen, daß die Steuerung über Tabellen ihre Vorteile hat.

15 Programmoptimierung in C

15.1 Möglichkeiten der Optimierung

In C kann man nach landläufiger Meinung schnellere Programme als in anderen Programmiersprachen schreiben. Diese Meinung ist schlicht und einfach falsch. Die Ablaufgeschwindigkeit eines durch einen C-Compiler erzeugten Programms ist nicht besser als die eines inhaltlich gleichen Programms, das in einer anderen Compilersprache geschrieben wurde. Voraussetzung ist natürlich ein guter Compiler mit einem guten Codeoptimierer.

Wahrscheinlich stammt diese Meinung aus der Zeit, als gute Optimierer unter den Compilern selten waren. Bei der Konzeption von C sind einige Spracheigenschaften festgelegt worden, die nur aus dem Mangel an guten Optimierern zu erklären sind, z.B. die Operatoren +=, -=, *=, /= etc. In der Tat erzeugen moderne Compiler für Befehle wie *a=a+17*; den gleichen Code wie für *a+=17*;. Auch die Operatoren ++ und -- sind daher zu erklären. Sie sind wohl der Ersatz für die Inkrement- und Dekrementbefehle, die in nahezu allen Assemblern verwendet werden können. Inzwischen erzeugen alle guten Compiler solche Befehle, selbst in Sprachen (wie Fortran und Pascal), die nicht über ähnliche Operatoren verfügen.

Dennoch gibt es einige Möglichkeiten in C, die Ihre Programme beschleunigen können. Als Beispiel lassen sich die Registervariablen und die Pointertechnik anführen. Die weitaus größten Reserven an Geschwindigkeit können Sie jedoch ausschöpfen, wenn Sie Ihr Programm sorgfältig analysieren und die zeitkritischen Stellen gezielt verbessern. Erfahrungsgemäß sind nur 5 - 10% eines Programms zeitkritisch, d.h. durch die Änderung einer kleinen Codestrecke erzielen Sie große Vorteile.

Die folgenden Überlegungen mischen C-spezifische Vorschläge und Vorschläge, die für alle Programmiersprachen gelten. Am Ende des Kapitels findet sich eine detaillierte Fallstudie, die Sie sich genau anschauen sollten.

15.2 Verwendung von Registervariablen

Die Verwendung der Speicherklasse *register* wirkt auf einigen Maschinen und mit einigen Compilern wahre Wunder – in anderen Konfigurationen ist der Effekt gleich Null. Warum dieses seltsame Bild? Ich habe schon erwähnt, daß die Angabe des Schlüsselworts *register* eine Empfehlung an den Compiler darstellt; er muß sich nicht an diese Empfehlung halten. Wenn z.B. die Zahl der verfügbaren Register erschöpft ist, hält er sich nicht an Ihre Empfehlung. Es gibt auch Compiler, die zwar das Schlüsselwort erlauben, aber in keinem Fall dazu zu bewegen sind, diese Empfehlung auch auszuführen. Andere Compiler legen – ob empfohlen oder nicht – immer einige Variablen in Registern ab. Es kann also dann passieren, daß durch das Schlüsselwort *register* das Programm nicht schneller wird, weil auch ohne dieses Schlüsselwort die Ablage in Registern ausgenutzt wurde.

Lattice-C und Aztec-C reagieren allerdings auf diese Empfehlung! Es ist dann sehr vom Programm abhängig, ob die Ablage in Registern einen Vorteil bringt. Das folgende Programm profitiert sicher nicht vom Schlüsselwort *register*:

```
char cfeld[100];
register int i;
  for (i=0; i < 1000; i++)
  {
    printf("\nEingabe:");
    gets(cfeld);
  }
```

Sie werden keinen meßbaren Unterschied zu einer Version feststellen, die auf *register* verzichtet, da der Geschwindigkeitsvorteil gering ist gegenüber der Ein-/Ausgabezeit.

Typische Kandidaten für *register*-Variablen sind Programme, die Schleifen mit hoher Durchlaufzahl und kurzem Schleifenkörper aufweisen. Der Schleifenkörper sollte keine peripheren Geräte ansprechen, die Wartezeiten auslösen können und keine Betriebssystemroutinen verwenden, deren Zeitverhalten undurchschaubar ist.

15.3 Großer und kleiner Code

Großer Code ist nicht immer schlechter Code und kleiner Code ist nicht immer guter Code – obwohl die Compilertests in den populären Zeitschriften etwas anderes verkünden. Kleiner Code wird nämlich nicht selten durch intensive Verwendung von Prozeduraufrufen erreicht. Prozeduraufrufe haben jedoch einen 'Overhead' dadurch, daß die Prozedurargumente auf den Stack gepackt werden müssen und daß in der gerufenen Prozedur Sicherungsmaßnahmen wie das Retten der Register getroffen werden müssen. Auf wirklich zeitkritischen Strecken (und nur dort!) ist deshalb der Ersatz von Prozeduren durch Makros zu empfehlen. Makros erzeugen mehr Code,

laufen aber schneller ab, da keine Argumentübergabe zur Laufzeit stattfindet. Auch ohne die Verwendung von Makros kann längerer Code zu einer Verbesserung der Durchlaufzeit führen.

16 Die ANSI-Norm für C

Derzeit wird eine Norm für C in den Normierungsgremien von ANSI vorbereitet. Der derzeitige Normentwurf trägt die Bezeichnung X3J11. C rückt damit in den Kreis der standardisierten Sprachen auf. Einer Sprache wie COBOL hat nicht zuletzt die Normierung das Überleben gesichert.

Von den geplanten Erweiterungen sind viele (z.B. die Typen *enum* und *void*) schon in aktuellen Compilern realisiert. Die Norm ist noch nicht verabschiedet (Stand 1987). Dennoch lassen sich die wichtigsten Änderungen gegenüber K&R schon aufzählen:

- Einführung des Typs *long double*. Damit wird die Rechengenauigkeit im Fließkommabereich erhöht.
- Einführung bzw. Sanktionierung der Typen *enum* und *void*. (Aufzählungstyp, leerer Funktionswert)
- Parallel zu *unsigned* gibt es ein Schlüsselwort *signed*. Alle Basistypen können *unsigned* und *signed* sein. Die Basistypen *long, char* und *short* können derzeit nach K&R nicht *unsigned* sein.
- Die Reihenfolge der Auswertung von Elementen eines komplexen Ausdrucks kann durch den einstelligen Operator + festgelegt werden.
- *float*-Arithmetik wird zugelassen. Bisher wurde bekanntlich alle Fließkommaarithmetik mit *double*-Werten ausgeführt. *float*-Arithmetik kann schneller sein als *double*-Arithmetik.
- Aufhebung der K&R-Restriktion bezüglich der Eindeutigkeit von Namen für Strukturelemente.
- Strukturen dürfen als Funktionsparameter übergeben werden. Funktionen dürfen Strukturen als Werte liefern.

- Zuweisungen von Strukturen werden ermöglicht. Bisher durften Strukturen nicht als Lwerte oder Rwerte erscheinen.
- Einführung des Konzepts 'Funktionsprototyp'. Funktionen können damit typmäßig und bezüglich der Zahl ihrer Argumente festgelegt werden. Die Einhaltung der Argumenttypen und der Zahl der Argumente wird zur Compilezeit geprüft. Es ist jedoch weiterhin möglich, Funktionen mit beliebig vielen Argumenten und unterschiedlichen Argumenttypen aufzurufen.
- Definition einer standardisierten C-Bibliothek.

Die Definition einer C-Standard-Bibliothek scheint mir das wichtigste Vorhaben dieser Norm zu sein. Damit wird die C-Bibliothek Bestandteil der Sprachdefinition.

Da die Verabschiedung dieser Norm noch nicht erfolgt ist, werden normkonforme Compiler noch auf sich warten lassen. Auch nach Verabschiedung der Norm dürfte es noch einige Zeit besser sein, die neuen Spracheigenschaften nicht auszunutzen. Schließlich wird es für einige Jahre noch alte C-Compiler im Feld geben. Wer also Programme transportieren will, sollte sich konservativ verhalten. Das fällt nicht schwer, da die meisten der vorgeschlagenen Spracherweiterungen im K&R-Rahmen nachgebildet werden können.

17 Eine kleine Sammlung von Beispielprogrammen

17.1 Konvertierroutinen

Das folgende Beispielprogramm stellt eine Sammlung von Konvertierroutinen dar, die auf jedem C-Compiler und auf jeder Maschine laufen sollten. Ich habe diese Routinen, die auf den ersten Blick überflüssig erscheinen mögen, geschrieben, weil sich gerade einige häufig benötigte Konvertierroutinen wie *scanf()* und *atoi()* auf unterschiedlichen Compilern unterschiedlich verhalten. Dies führt soweit, daß einige getestete Programme aus dem Public-Domain-Lager partout nicht laufen wollen. Die folgenden Beispielsroutinen verwenden nur C-Standardsprachmittel und sind deshalb transportabel. Ein zweiter Grund kam hinzu: selbst für einfachste Programme wird mitunter riesiger Code erzeugt, wenn *printf()* aufgerufen wird. Zumindest für eine Anwendung kann man sich aber riesigen Code nicht erlauben: beim Schreiben von ROM-residenten Programmen. Diese Konvertierroutinen bilden also den Kern für ein eigenes, abgespecktes *printf()*. Die interne Logik von *printf()* habe ich im Kapitel 10 gezeigt; dort fehlen allerdings die Konvertierroutinen, die hier nachgereicht werden. Ein Ersatz für *scanf()* ist ebenso einfach herstellbar. Die Prozedur *streval()* könnte neben *myatoi* in einem eigenen *scanf()* verwendet werden. Das Hauptprogramm *main()* stellt nur einen einfachen Testrahmen für die Routinen bereit.

Es gibt verschiedene Ansätze und Verfahren, Konvertierroutinen zu schreiben. Das bekannteste Verfahren zur Umwandlung von Binärmustern in ASCII-Strings bildet das Divisions-Rest-Verfahren, das ich hier in *intdec()* und *inthex()* verwende. Da dieses Verfahren jedoch auf der Division beruht, ist es nicht immer sehr effizient. Deshalb verwende ich in der Routine *intbin()* ein Verfahren, das den Shiftoperator einsetzt. Dieses wesentlich effizientere Verfahren könnte auch für *inthex()* verwendet werden; es bietet sich immer dann an, wenn das Bitmuster der internen Ablage direkt

interpretiert werden kann, also bei der Konvertierung in binäre, oktale und hexadezimale Darstellungen.

Die Routine *myatoi()* konvertiert aus einem String in ein Binärmuster. Mit nur wenigen Abänderungen können aus dieser Prozedur Konvertierungen für Oktal- und Hexadezimalstrings abgeleitet werden.

Etwas komplizierter ist die Prozedur *streval(),* die aus C-Stringkonstanten die interne Ablage in einem *char*-Array erzeugt. So oder ähnlich können Sie sich die Arbeitsweise des Compilers vorstellen, wenn er auf einen C-Konstantstring trifft. Die Routine erwartet C-Strings zwischen doppelten Hochkommata; im String dürfen die beiden offiziellen Ersatzdarstellungen für *char*-Konstanten (s. Kap. 5) verwendet werden.

Da ich generell versuche, transportable Programme zu schreiben, erspart diese Art von Prozeduren unnötige Arbeit und Ärger beim Übergang von einer Maschine zur anderen. Bitte beachten Sie, daß Routinen für die Konvertierung von Fließkommazahlen aufwendiger sind und zudem nicht transportabel gehalten werden können, weil nicht alle Compiler die IEEE-Darstellung für Fließkommazahlen verwenden. Je nach verwendetem Compilertyp müssen einige Konstanten umdefiniert werden. Eine generelle automatische Anpassung (in *intbin()* skizziert) an die jeweils gültige Wortgröße (16 Bit oder 32 Bit) wäre zwar möglich aber ineffizient, da die Prozeduren u.U. unter zeitkritischen Bedingungen ablaufen.

```
/****************************************

                  Konvert
            last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

Eine Sammlung von Konvertierungsroutinen

****************************************/

#include <stdio.h>

#define MAX10  10000
#define MAX16  0x1000

/* Dezimalkonvertierung einer int-Zahl in einen String */
/* Verfahren: Divisions-Rest-Verfahren                 */
/* Vorlaufende Nullen werden unterdrueckt              */
char *intdec(zahl)
int zahl;
{
  static char ergebnis[10];
  int maxpot,flag;
  char c,*dest;
```

```
  flag   = 0;          /* Flag Ziffer abgelegt   */
  maxpot = MAX10;      /* Wert 10 hoch n         */
  dest   = ergebnis;   /* Pointer Ergebnisstring */
  if (zahl < 0)
    {
     zahl *= (-1);
     *dest++ = '-';
    }
  while (maxpot >= 1)
    {
     c = zahl / maxpot + '0';
     if (( c != '0') || ( maxpot == 1) || flag)
       {
        flag    = 1;
        *dest++ = c;
       }
     zahl   %= maxpot;
     maxpot /= 10;
    }
  *dest = 0;
  return(ergebnis);
}   /* end intdec */

/* Hexadezimalkonvertierung einer int-Zahl in einen String  */
/* Verfahren: Divisions-Rest-Verfahren                      */
/* Im Unterschied zu intdec wird nie ein Vorzeichen erzeugt */
/* Verfahren ist durch Links-Shiften in Vierergruppen       */
/* ersetzbar                                                */
/* Vorlaufende Nullen werden unterdrueckt                   */
char *inthex(zahl)
unsigned zahl;
{
  static char ergebnis[10];
  unsigned maxpot,flag;
  char c,*dest;

  flag   = 0;          /* Flag Ziffer abgelegt   */
  maxpot = MAX16;      /* Wert 16 hoch n         */
  dest   = ergebnis;   /* Pointer Ergebnisstring */
  while (maxpot >= 1)
    {
     c = zahl / maxpot + '0';
     if (( c != '0') || ( maxpot == 1) || flag)
       {
        flag    = 1;
        if (c > '9') c += 7;
        *dest++ = c;
       }
     zahl   %= maxpot;
     maxpot /= 16;
    }
  *dest = 0;
  return(ergebnis);
}   /* end inthex */
```

```
/* Binaerkonvertierung einer int-Zahl in einen String        */
/* Verfahren: Links Shiften                                   */
/* Im Unterschied zu intdec wird nie ein Vorzeichen erzeugt */
/* Vorlaufende Nullen werden unterdrueckt                    */
char *intbin(zahl)
unsigned zahl;
{
  static char ergebnis[33];
  int flag,n,muster,nbits;
  char *dest;

  flag = 0;
  dest = ergebnis;
  nbits = sizeof(unsigned) * 8;
  if (nbits == 16) muster = 0x8000;
  else             muster = 0x80000000;

  for (n = 0; n < nbits; n++)
    {
     if (zahl & muster)
       {
        flag     = 1;
        *dest++ = '1';
       }
     else
       {
        if (flag) *dest++ = '0';
       }
     zahl = zahl << 1;
    }
  if (! flag) *dest++ = '0';
  *dest = 0;
  return(ergebnis);
}   /* end intbin */
/* Konvertierung ASCII-Dez.String nach int-Zahl */
/* Liefert -1 im Fehlerfall                     */
/* Liefert sonst die Zahl der konv. Stellen     */
int myatoi(str,iadr)
char *str;
int *iadr;
{
  int nospace,c;
  int zahl,vorz,stellen;

  nospace = 0;   /* flag Zeichen <> white space */
  zahl    = 0;
  vorz    = 1;
  stellen = 0;
  while (*str != 0)
    {
     c = *str++ & 0xff;
     stellen++;
     if ((c == ' ') || (c == 0x0a) ||
         (c == 0x09) || (c == 0x0d))
```

```
        {
         if (nospace) break;
         continue;
        }
      if ((c == '-') && (! nospace))
        {
         vorz = -1;
         continue;
        }
      if ((c >= '0') && (c <= '9'))
        {
         zahl = (zahl * 10) + (c - '0');
         nospace = 1;
        }
      else return(-1);
    }
  *iadr = zahl * vorz;
  return(stellen);
}   /* end myatoi */

/* Stringevaluierung                                  */
/* Erkennt \n,\r,\t,\b und \777 (Oktalzahlen)         */
/* liefert die Zahl der konv. Stellen                 */
/* liefert -1 im Fehlerfall                           */
int streval(inpstr,outstr)
char inpstr[],*outstr;
{
  int n,nhk,zahl,esc,linp;
  char c;

  nhk     = 0;  /* zaehlt die Hochkommata */
  esc     = 0;  /* Flag in Ersatzdarstellung */
  linp    = strlen(inpstr);
  *outstr = 0;

  for (n=0; n < linp; n++)
    {
     c = inpstr[n];
if (c == 0x0a) break;
/* Alles vor '"' ignorieren */
if ((c != '"') && (nhk == 0)) continue;
if (esc)
  {
   if ((c >= '0') && (c <= '7'))
     {
      /* in einer Oktalzahl: konvertieren       */
      /* Maximal drei Stellen beruecksichtigen */
      zahl = (zahl * 8) + (c - '0');
      /* naechstes Zeichen ansehen */
      c = inpstr[n+1];
      if (! ((c >= '0') && (c <= '7')))
        {
         /* Ende der Oktalzahl erreicht */
         *outstr++ = zahl;
         esc = 0;
         continue;
```

```
        }
      if (esc++ >= 3)
        {
         /* Drei Stellen bearbeitet */
         *outstr++ = zahl;
         esc = 0;
         continue;
        }
      continue;
     }   /* if ((c >= '0') && (c <= '7')) */
   else
     {
      /* sonstige \ - Sequenz                    */
      /* das naechste Zeichen wird direkt oder */
      /* interpretiert uebernommen               */
      if (c == 'n') c = 0x0a;   /* \n LF  */
      if (c == 'b') c = 0x08;   /* \b Bsp */
      if (c == 't') c = 0x09;   /* \t Tab */
      if (c == 'r') c = 0x0d;   /* \r CR  */
      *outstr++ = c;
      esc = 0;
      continue;
     }
  }
if (c == '"')
  {
   nhk++;
   if (nhk >= 2)
     {
      /* Ende eines Strings */
      *outstr = 0;
      return(n+1);
     }
   continue;
  }
if (c == '\\')
  {
   /* Start einer Escapesequenz */
   esc  = 1;
   zahl = 0;
   continue;
       }
     /* Normales Zeichen : wird uebernommen */
     *outstr++ = c;
    }
  return(-1);
}   /* end streval */

main()
{
  char cfeld[50],ofeld[50];
  int ivar,n;

  printf("\n%s",intdec(127));
  printf("\n%s",intdec(-127));
  printf("\n%s",inthex(127));
```

```
  printf("\n%s",inthex(-127));
  printf("\n%s",intbin(127));
  printf("\n%s",intbin(-127));
  printf("\nGib Dezimalzahl: ");
  gets(cfeld);
  if (myatoi(cfeld,&ivar) < 0) printf("\nKonvertierfehler");
  else printf("\nKonvertiert: %d",ivar);
  printf("\nGib String zwischen Hochkommata:");
  gets(cfeld);
  ivar = streval(cfeld,ofeld);
  if (ivar < 0) printf("\nKonvertierfehler");
  else
    {
     printf("\n");
     ivar = strlen(ofeld);
     for (n=0; n <= ivar; n++)
       printf("%02x ",ofeld[n]);
    }
}   /* end main */
```

17.2 C im kommerziellen Einsatz: DIN-gerechte Sortierung

Das folgende Programm *dinsort.c* ist eine praktische Anwendung von C im kommerziellen Bereich. C steht bekanntlich im Ruf, nur für technische Anwendungen wirklich brauchbar zu sein. Dieses Programm soll das Gegenteil beweisen.

Das Programm *dinsort.c* soll eine auf das Deutsche zugeschnittene normgerechte Sortierung erzeugen. Sie kennen sicher das Problem: selbst renommierte Datenbanksysteme verfügen bisweilen nur über eine ASCII-Sortierung; die deutschen Umlaute werden hoffnungslos falsch einsortiert. Genau dies macht das Programm (hoffentlich) korrekt. Darüber hinaus erzeugt es ein festes Sortierformat aus einem variablen Eingabeformat. Ein variables Satzformat kann schlecht sortiert werden. Deshalb erzeuge ich aus dem variablen Format ein festes Format, das ohne Klimmzüge sortiert werden kann. Sie sollten sich vorstellen, daß feste Strukturen sortiert werden, an denen ein Pointer auf den variablen Inputsatz 'klebt'.

Die Umlaute werden hier über eine pflegbare Tabelle konvertiert. Diese Konvertierungstabelle kann auch für andere Zwecke konvertiert werden. Einige kommerzielle Anforderungen (z.B. Sortierung von Telefonbüchern) verlangen z.B., daß Sonderzeichen wie '&' (kaufmännisches 'und') ausbuchstabiert werden; andere Anforderungen wiederum wollen Groß- und Kleinbuchstaben gleich behandelt wissen. Das alles kann leicht über diese Tabelle erfolgen. Für die Konvertierung wird ein Buchstabenbaum verwendet, der vielleicht nicht einfach zu verstehen ist. Die Konvertierung erlaubt n:m Umsetzung, d.h. Umsetzung von beliebig langen Strings in beliebig lange Strings. Allerdings verwende ich hier nicht die Stringauswertungsroutine *streval()* (s. obige Konvertierungsroutinen), die sicher universeller wäre.

Die zweite pflegbare Tabelle nenne ich 'Feldbeschreibungstabelle'. Sortiert werden normalerweise 'Datensätze'; darunter können Sie sich Strukturen vorstellen, deren Elemente feste Länge haben und 'Felder' genannt werden. Die Feldbeschreibungstabelle gibt an, wo die extrahierten Felder im Datensatz abgelegt werden sollen, wie lang sie sind und durch welchen Trenner (Dezimalcode angeben!) sie vom rechts danebenstehenden Feld abgetrennt werden. Der Sinn dieser Tabelle liegt erstens darin, die Suche nach den Feldinhalten zu steuern; Feldinhalte werden hier aus Texten extrahiert, die einen ziemlich variablen Aufbau haben. Normalerweise geht man in der kommerziellen Datenverarbeitung eher von einem festen Aufbau aus. Die Tabelle steuert zweitens den Aufbau des Sortiersatzes. Die Sortiersätze haben natürlich einen festen Aufbau. Vorlaufende Blanks in Feldinhalten werden ignoriert. Ich habe hier der Einfachkeit halber nicht alle Möglichkeiten dieses Konzept ausgenützt: Sie könnten z.B. das Programm so ändern, daß eine Steuerung des Hauptsortierkriteriums möglich ist, d.h. des Feldes, das den Hauptsortierschlüssel bildet.

Die Sortierung geschieht beim Einlesen der Inputsätze; es gibt also keinen eigenen Sortierlauf. Sortiert werden nur Pointer – keine Satzinhalte! Zum Sortieren wird eine Struktur namens *RECORD* verwendet, die einen Pointer auf den Originalinputsatz (variable Felder) und auf den modifizierten Inputsatz (feste Felder) enthält. Die Sortierstrukturen werden als verkettete Liste abgelegt; zur Beschleunigung der Sortierung dient die Stützpunkttabelle *hash*, die ihren Namen nicht ganz zu Recht trägt.

Das Hauptprogramm *main()* zeigt die Struktur des Gesamtprogramms: in *rdfilter()* wird die Filtertabelle (Konvertierungstabelle) eingelesen, in *rdfdescript()* die Feldbeschreibungstabelle. Dann werden die variablen Inputsätze eingelesen, in ein festes Datenformat gebracht *(fixrecord())* und einsortiert *(sortit())*. Schließlich werden die sortierten Datensätze wieder ausgegeben, wobei nur der Originalsatz ausgegeben wird, der ja am Sortiersatz 'klebt'.

Wenn Sie Probleme haben mit Ihren Tabellen: beim Aufruf mit der Option *-t* werden zusätzliche Kontrollausdrucke erzeugt, die die Fehlersuche erleichtern sollen.

Das Programm verwendet Module, die früher schon in leicht abgewandelter Form aufgetaucht sind.

Zunächst einige Beispiele für die Daten, die *dinsort.c* erwartet.

Eine Konvertierungstabelle. Das Zeichen '=' dient als Trenner:

```
ä=ae
Ä=Ae
ö=oe
Ö=Oe
ü=ue
Ü=Ue
ß=ss
```

Eine Feldbeschreibungstabelle. Jedem Feld entspricht eine Zeile. Pro Feld wird der Feldanfang (gezählt ab Null), die maximale Feldlänge und der Dezimalcode des Trenners eingetragen:

```
0 20 44
21 20 59
41 4 32
45 20 10
```

Ein typischer Inputfile (bitte beachten Sie die vorlaufenden Blanks):

```
 Huckert, Edgar; 6641 Harlingen
Dacken,Gisela; 402 Neuß
   Abraham,Fritz; 5542 Delfingen
Ährlich,Peter; 4021 Moskau
```

Der erzeugte Output (die seltsame Eingabe mit vorlaufenden Blanks bleibt absichtlich erhalten!):

```
   Abraham,Fritz; 5542 Delfingen
Ährlich,Peter; 4021 Moskau
Dacken,Gisela; 402 Neuß
 Huckert, Edgar; 6641 Harlingen
```

```
/****************************************

                  DIN-Sort
            last update 10/07/87
      AMIGA-Version by Frank Kremser
PC-Original-Version by Dr. Edgar Huckert
       (C) 1987  by Markt & Technik

****************************************

DIN - aehnliche Sortierung fuer deutschsprachige Eingaben
verwendet eine n:m Konvertierung fuer die Erzeugung des
DIN - Sortiercodes
erzeugt zum Sortieren eine definierbare Fix-Record Struktur
ignoriert vorlaufende Blanks in Feldinhalten

****************************************/

#include <stdio.h>

#define CHAR unsigned char
#define LF 0x0a

/* Struktur erweiterter Datensatz */
typedef struct record
{
  CHAR *orecord;          /* Originalrecord     */
  CHAR *frecord;          /* modif. record      */
  struct record *next;    /* Nachfolgerpointer */
} RECORD;
```

```
/* Hashtabelle für Sortierung */
RECORD *hash[256];
int test,nfeld;
char *fgets(),*malloc();

/* Feldbeschreibungstabelle        */
/* 3 Eintraege pro Feld:            */
/* 1: Startposition Ablage         */
/* 2: Laenge Ablage                 */
/* 3: Trenner zum naechsten Feld   */
#define MAXNFELD  10
int fdescript[MAXNFELD][3];

/* Struktur eines Knotens des Buchstabenbaumes */
/* fuer n:m Konvertierung                      */
typedef struct bbaum
{
  CHAR zei;              /* Zeicheneintrag */
  struct bbaum *sohn;    /* Zeiger auf ersten Sohn */
  struct bbaum *bruder;  /* Zeiger auf Bruder */
  CHAR *ergebnis;        /* Zeiger auf Resultatsstring */
}   BBAUM;
/* Verzeichnis der Teilbaumanfaenge */
BBAUM *bbarray[256];

/* Abbruchroutine: gibt Meldung aus und geht */
/* ins Betriebssystem zurueck                */
abbruch(meld)
CHAR *meld;
{
  printf("\nAbbruch: %s ---\n",meld);
  exit(1);
}   /* end abbruch */

/* Einen Knoten des Buchstabenbaums anlegen */
BBAUM *make_node(c)
CHAR c;
{
  BBAUM *anode;

  anode = malloc(sizeof(BBAUM));
  anode->zei      = c;
  anode->sohn     = NULL;
  anode->bruder   = NULL;
  anode->ergebnis = NULL;
  return(anode);
}   /* end make_node */

/* Prueft vom Vaterknoten vater ausgehend, ob ein */
/* Kindknoten mit Zeicheneintrag c existiert      */
BBAUM *get_child(c,vater)
CHAR c;
BBAUM *vater;
{
  BBAUM *anode;
```

```
  if (vater == NULL) return(NULL);
  anode = vater->sohn;
  if (anode == NULL) return(NULL);
  if (anode->zei == c) return(anode);
  anode = anode->bruder;
  while (anode != NULL)
    {
     if (anode->zei == c) return(anode);
     anode = anode->bruder;
    }
  return(NULL);
}   /* end get_child */

/* Anlegen eines Kindknotens unter dem Vaterknoten vater */
BBAUM *put_child(c,vater)
CHAR c;
BBAUM *vater;
{
  BBAUM *anode,*lastchild;

  anode = vater->sohn;
  if (anode == NULL)
     /* Sohn - Pointer noch nicht besetzt */
     anode = vater->sohn = make_node(c);
  else
    {
     /* Bis zum letzten Kind laufen */
     while (anode != NULL)
       {
        lastchild = anode;
        anode = anode->bruder;
       }
     lastchild->bruder = anode = make_node(c);
    }
  return(anode);
}   /* end put_child */

/* prueft, ob ein Knoten zu diesem Zeichen schon existiert. */
/* Falls nicht: Neuanlegen eines Knotens                    */
BBAUM *ins_node(c,lnode)
CHAR c;
BBAUM *lnode;
{
  BBAUM *anode;

  if (lnode == NULL)
    {
     /* Anfangsknoten */
     if (bbarray[c] == NULL)
       anode = bbarray[c] = make_node(c);
     else anode = bbarray[c];
    }
  else
    {
```

```
      /* Kein Anfangsknoten */
      anode = get_child(c,lnode);
      if (anode == NULL) anode = put_child(c,lnode);
     }
  return(anode);
}   /* end ins_node */

/* Einlesen des Filterfiles und Umwandlung in einen */
/* Buchstabenbaum                                   */
rdfilter()
{
  int i,n,zlen,part,zbufl;
  CHAR *result,filna[60];
  BBAUM *actnode;
  CHAR zbuf[80],zeile[80];
  FILE *ffil;

  /* bbarray initialisieren */
  for (n = 0; n < 256; n++)
    bbarray[n] = NULL;

  printf("\nFiltertabelle:");
  gets(filna);
  ffil = fopen(filna,"r");
  if (ffil == NULL) abbruch("Filtertabelle nicht gefunden");
  while (1)
    {
     if (fgets(zeile,80,ffil) == NULL) break;
     /* printf("\n%s",zeile); */
     zlen    = strlen(zeile);
     /* Leerzeilen ignorieren */
     if (zlen <= 0)       continue;
     if (zeile[0] == LF) continue;
     part    = 1;
     actnode = NULL;
     zbufl   = 0;
     for (i=0; i < zlen; i++)
       {
        if (zeile[i] == LF) break;
        if (zeile[i] == '=')
          {
           part = 2;
           if (actnode == NULL) break;
           continue;
          }
        if (part == 1)
          {
           /* Suchstring im Buchstabenbaum ablegen */
           actnode = ins_node(zeile[i],actnode);
          }   /* if part == 1 */
        else
          {
           /* Resultatsstring im Zwischenpuffer ablegen */
           zbuf[zbufl++] = zeile[i];
           zbuf[zbufl]   = 0;
          }
```

```
        }   /* for i=0; ... */
      if (part != 2)
        {
         printf("\n%s",zeile);
         abbruch(" Fehler im Filtereintrag ");
        }
      /* Jetzt den exakt dimensionierten Ergebnisstring anlegen */
      actnode->ergebnis = malloc(strlen(zbuf) + 1);
      strcpy(actnode->ergebnis,zbuf);
     }   /* while 1 */
  fclose(ffil);
}   /* end rdfilter */

/* Die Feldbeschreibungstabelle lesen und in  */
/* fdescript ablegen.                         */
/* Rueckgabe: Anzahl der Felder               */
rdfdescript()
{
  FILE *tfil;
  char zeile[60];
  int nfeld;

  nfeld = 0;
  printf("\nFeldbeschreibungstabelle:");
  gets(zeile);
  if ((tfil = fopen(zeile,"r")) == NULL)
    abbruch("Feldbeschreibungstabelle nicht gefunden");
  while (1)
    {
     if (fscanf(tfil,"%d %d %d",&fdescript[nfeld][0],
       &fdescript[nfeld][1],&fdescript[nfeld][2]) < 3) break;
     if (++nfeld > MAXNFELD) abbruch("Zu viele Felder");
    }
  if (test) printf("\nAnzahl der Felder: %d",nfeld);
  fclose(tfil);
  return(nfeld);
}   /* end rdfdescript */

/* prueft, ob ein Knoten im Buchstabenbaum fuer das */
/* Inputzeichen c vorliegt                          */
BBAUM *identify(c,vater)
CHAR c;
BBAUM *vater;
{
  if (vater == NULL) return(bbarray[c]);
  return(get_child(c,vater));
}   /* end identify */

/* Liest das naechste Zeichem vom Inputfile und  */
/* prueft es anhand des Buchstabenbaums          */
/* Rueckgabe: Pointer auf Filterergebnis         */
/* Vorsicht: Prozedur ist rekursiv!              */
CHAR *filter(pcadr,lastnode)
CHAR *pcadr;
BBAUM *lastnode;
```

```
{
  int ic;
  BBAUM *actnode;
  static CHAR ergebnis[20];
  static int ierg;

  if (lastnode == NULL)
    {
     ierg         = 0;
     ergebnis[0] = 0;
    }
  ic = *pcadr;
  if (ic <= 0) return(ergebnis);
  if (ierg < (sizeof(ergebnis) - 1))
    {
     ergebnis[ierg++] = ic;
     ergebnis[ierg]   = 0;
     /* Naechsten Knoten im Buchstabenbaum identifizieren */
     actnode = identify(ic,lastnode);
    }
  else return(ergebnis);
  /*
     printf("\nic=%c actnode=%x lastnode=%x",
            ic,actnode,lastnode);
  */
  if (actnode == NULL)
    /* Kein String identifiziert */
    return(ergebnis);
  if (actnode->ergebnis != NULL)
    /* String identifiziert */
    return(actnode->ergebnis);
  /* Mitten in String: weitermachen */
  return(filter(pcadr + 1));
}   /* end filter */

/* liefert Pointer auf den Eintrag vor dem */
/* Einfuegepunkt in einer Kette            */
RECORD *plentry(str)
unsigned char *str;
{
  RECORD *pchain,*plast;

  pchain = hash[*str];
  plast  = NULL;
  while (pchain != NULL)
    {
     if (strcmp(pchain->frecord,str) >= 0) break;
     plast  = pchain;
     pchain = pchain->next;
    }
  return(plast);
}   /* end plentry */
```

```
/* Einen Record richtig einsortieren */
sortit(arec)
RECORD *arec;
{
  int fc;
  RECORD *plast;

  fc = *(arec->frecord);
  if (hash[fc] == NULL)
    {
     /* Noch kein Hash-Eintrag da */
     hash[fc] = arec;
     return;
    }
  /* Adresse des vorhergehenden Eintrags ermitteln */
  plast = plentry(arec->frecord);
  if (plast != NULL)
    {
     /* Mitten in Kette einhaengen */
     arec->next  = plast->next;
     plast->next = arec;
    }
  else
    {
     /* An den Anfang einer Kette haengen */
     arec->next = hash[fc];
     hash[fc] = arec;
    }
}   /* end sortit */

/* Legt aus dem Originalrecord orec einen record mit */
/* fester Struktur an, der sortierfaehig ist         */
/* benutzt Array fdescript                           */
/* Rueckgabe: NULL im Fehlerfall - sonst Adresse des */
/* angelegten Records mit festen Feldgrenzen         */
CHAR *fixrecord(orec)
CHAR *orec;
{
  CHAR *frec,*hadr,*src,*dest,modrec[200];
  int n,fnum,frgroesse,actlen,maxlen;

  n = nfeld - 1;
  frgroesse = fdescript[n][0] + fdescript[n][1];
  frec = (CHAR *)malloc(frgroesse+1);
  if (frec == NULL) abbruch("kein Speicher");
  /* Felder des festen records mit Blanks vorbesetzen */
  hadr = frec;
  for (n=0; n < frgroesse; n++)
    *hadr++ = ' ';
    *hadr = 0;
    /* Sortierfilter auf Originalrecord anwenden    */
    /* erzeugt wird der modifizierte Originalrecord */
    dest = modrec;
    src  = orec;
    while (*src != 0)
```

```
    {
     hadr = filter(src,NULL);
     while (*hadr != 0)
       *dest++ = *hadr++;
     src++;
    }
  *dest = 0;
  if (test)
    printf("\nmodifizierter Record: %s",modrec);
  /* Jetzt aus dem modifizierten Originalrecord einen */
  /* record mit festen Feldgrenzen erzeugen           */
  src     = modrec;
  fnum    = 0;     /* Index in fdescript */
  dest    = frec + fdescript[fnum][0];
  maxlen  = fdescript[fnum][1];
  actlen  = 0;
  while (*src != 0)
    {
     if ((*src == fdescript[fnum][2]) && (actlen >= 1))
       {
        /* Trenner identifiziert */
        fnum++;
        if (test)
          printf("\nfnum=%d Trenner=%x %c", fnum,*src,*src);
        if (fnum >= nfeld) break;
        src++;
        dest    = frec + fdescript[fnum][0];
        maxlen  = fdescript[fnum][1];
        actlen  = 0;
        continue;
       }
     if (maxlen > 0)
       {
        /* Vorlaufende Blanks ignorieren */
        if (! ((*src == ' ') && (actlen == 0)))
          {
           /* Feldinhalt ablegen */
           actlen++;
           maxlen--;
           *dest++ = *src;
          }
       }    /* if maxlen > 0 */
     src++;
    }     /* while *src != 0 */
  if (fnum < nfeld) abbruch("zu wenig Felder");
  if (test) printf("\nfixrecord=%s",frec);
  return(frec);
}    /* end fixrecord */

/* naechsten Originalrecord lesen, modifizieren und */
/* einsortieren                                     */
RECORD *nextrecord(ifil)
FILE *ifil;
{
  RECORD *arec;
  CHAR orec[200],*memrec;
  int lrec;
```

```
  while (1)
    {
     if (fgets(orec,200,ifil) == NULL) return(NULL);
     if (orec[0] == LF) continue;
     break;
    }
  lrec = strlen(orec);
  memrec = (CHAR *)malloc(lrec);
  if (memrec == NULL) abbruch("Kein Speicher");
  strcpy(memrec,orec);
  arec = (RECORD *)malloc(sizeof(RECORD));
  if (arec == NULL)   abbruch("Kein Speicher");
  /* Pointer auf Nachfolger ablegen */
  arec->next     = NULL;
  /* Adresse auf Originalrecord ablegen */
  arec->orecord = memrec;
  /* Record mit festen Feldern erzeugen */
  arec->frecord = fixrecord(orec);
  if (arec->frecord == NULL) abbruch("Fehler bei Aufbereitung");
  /* Record einsortieren */
  sortit(arec);
  return(arec);
}   /* end nextrecord */

main(argc,argv)
int argc;
char *argv[];
{
  int n,lil;
  FILE *ifil,*ofil;
  CHAR filna[60];
  RECORD *pchain;

  test = 0;
  if (argc > 1)
    {
     if (strcmp(argv[1],"-t") == 0) test = 1;
    }
  /* Hashtabelle fuer Datensaetze initialisieren */
  for (n=0; n < 256; n++)
    hash[n] = NULL;

  /* Filterfile einlesen */
  rdfilter();

  /* Feldbeschreibungstabelle lesen */
  nfeld = rdfdescript();

  printf("\nInputfile:");
  gets(filna);
  ifil = fopen(filna,"r");
  if (ifil == NULL) abbruch("Inputfile nicht gefunden");

  printf("\nOutputfile:");
  gets(filna);
  ofil = fopen(filna,"w");
  if (ofil == NULL) abbruch("Outputfile nicht angelegt");
```

```
  /* Inputrecords zeilenweise verarbeiten */
  /* Dabei konvertieren und sortieren     */
  while (1)
    {
     if (nextrecord(ifil) == NULL) break;
    }

  /* Jetzt die sortierten und aufbereiteten Records */
  /* wieder ausgeben (nur Originalrecord)           */
  printf("\n");
  for (n=0; n < 256; n++)
    {
     if (hash[n] == NULL) continue;
     pchain = hash[n];
     while (pchain != NULL)
       {
        if (fprintf(ofil,"%s",pchain->orecord) < 0) goto ende;
        printf("%s",pchain->orecord);
        pchain = pchain->next;
       }
    }
ende:
  fclose(ifil);
  fclose(ofil);
  exit(0);
}   /* end main */
```

18 Kleines Glossar

Abgeleiteter Datentyp: nicht zusammengesetzter Datentyp, der aus einem Basistyp abgeleitet ist. Pointer und Funktionen sind abgeleitete Datentypen.

Argument: ein Wert, der zur Steuerung (Parametrisierung) einer Prozedur verwendet wird. Man unterscheidet aktuelle und formale Argumente. Argumente werden in C durch "call by value" auf dem Stack übergeben.

Array: eine Folge von C-Objekten des gleichen Datentyps. Deutsche Bezeichnung: Feld.

Attribut: Eigenschaft eines C-Objekts wie Datentyp, Gültigkeitsbereich, Speicherklasse.

Ausdruck: Basiselement von C-Sprachkonstrukten. Ausdrücke lassen sich durch Operatoren zu neuen Ausdrücken kombinieren. Jeder Ausdruck kann in C als Befehl verwendet werden.

Basisdatentyp: einfacher Datentyp wie *char, unsigned char, unsigned, int, short, long, double, float.*

Basisroutinen: die grundlegenden IO-Routinen der C-Bibliothek, auf denen die Standardroutinen aufsetzen. Sie verwenden Dateinummern (file descriptors) statt Filepointern. Zu ihnen gehören *open(), creat(), read(), write()* etc. Sie werden auch UNIX-IO-Routinen genannt.

Befehl: jeder C-Ausdruck ist ein gültiger Befehl. Syntaktisch korrekte Kombinationen aus Ausdrücken und C-Schlüsselwörtern sind Befehle. Es gibt in C einfache Befehle und zusammengesetzte Befehle (Blöcke).

Binder: deutscher Ausdruck für Linker.

Bit: die kleinste Informationseinheit in digitalen Rechenanlagen.

Block: eine Folge von C-Befehlen zwischen geschweiften Klammern.

Byte: eine Folge von 8 Bits, meist zur Zeichendarstellung verwendet. Der Basisdatentyp *char* belegt ein Byte.

Bibliothek: eine Sammlung von Funktionen, die zusammen mit einem C-Compiler geliefert werden und dafür sorgen, daß C-Programme in der jeweiligen Maschinen- und Betriebssystemumgebung ablauffähig sind. Die Ein-/Ausgabe wird in C vollständig über Bibliotheksfunktionen abgewickelt.

Call by adress: s. Call by reference

Call by descriptor: in einigen Programmiersprachen üblicher Übergabemechanismus, bei dem eine strukturartige Beschreibung eines Prozedurarguments übergeben wird. In C nicht vorgesehen, aber leicht erzielbar.

Call by reference: Übergabe der Adresse eines Arguments an eine Prozedur. In C durch den Adreßoperator & erzielbar.

Call by value: Übergabe des Wertes eines Arguments an eine Prozedur.

Datei: deutscher Name für File.

Debugger: ein Programm, mit dem Programme zwecks Fehlersuche gestartet werden können. Erlaubt schrittweise Ausführung von Programmen auf CPU-Befehlsebene oder C-Befehlsebene, Kontrolle des Speichers, Setzen von Unterbrecherpunkten etc. Symbolische Debugger können die Symboltabellen eines Linkers verarbeiten (z.B. SYMDEB, CODEVIEW).

Definition: Festlegung von Konstanten und Makros für den Präprozessor, Deklaration und Befehlsfolge einer Prozedur.

Deklaration: Festlegen des Namens und der Attribute eines C-Objekts.

Dekrementieren: Erniedrigen

Directory: Verzeichnis (Katalog) einer Menge von Dateien.

Einfacher Datentyp: s. Basisdatentyp.

Extern: ein externes C-Objekt ist außerhalb des Blocks definiert, zu dem es extern ist. Sein Name und seine Attribute sind jedoch innerhalb des Blocks bekannt.

File: Eine Folge von Bytes, die auf einem Massenspeicher abgelegt ist und unter einen Filenamen (Benutzersicht) oder unter einer Filenummer bzw. einem Filepointer (C-Sicht) erreichbar sind. Andere Denkweisen als das Konzept "Folge von Bytes" (Gliederung in Records, Indexfiles etc.) sind in C auf der untersten Ebene nicht üblich, lassen sich aber simulieren.

Fließkommazahl: Zahl mit einem ganzzahligen und einem gebrochenen Anteil. Interne Darstellung in C-Compilern meist im IEEE-Format. Moderne Maschinen haben eventuell spezielle Coprozessoren für die Verarbeitung von Fließkommabefehlen (z.B.8087 von Intel oder 68881 von Motorola).

Funktion: eine Prozedur, die einen Wert liefert. In C sind alle Prozeduren Funktionen, müssen aber nicht als solche verwendet werden.

Global: globale Daten sind in allen Prozeduren eines Programms zugänglich. Sie sind deshalb extern zu allen Programmodulen.

Gültigkeitsbereich: die Codestrecke, über die der Name und die Attribute eines C-Objekts bekannt sind.

Hauptprogramm: in C muß das Hauptprogramm den Prozedurnamen *main* haben. *main()* definiert den Anfangspunkt eines Programms beim Ablaufen.

Inkrementieren: Erhöhen.

Integer: ganze Zahl

Intern: ein internes C-Objekt ist nur innerhalb des Blocks bekannt, in dem das Objekt deklariert wurde.

Interrupt: Ein asynchrones Ereignis, das die Unterbrechung des laufenden Programms verursacht und in einer Interruptserviceroutine behandelt wird. Wichtiges Konzept beim Schreiben von Treibern.

K&R: Abkürzung für Kernighan und Ritchie. Diese beiden Autoren haben das Standardwerk zur Definition von C geschrieben. Ritchie gilt zudem als Miterfinder von UNIX. Referenz: Kernighan, B.W., Ritchie, D.M.: The C Programming Language, Englewood Cliffs: Prentice Hall (1978)

Kommandoprozessor: Ein Programm, das den Kontakt zwischen Anwender und Betriebssystem herstellt, indem es die Ausführung von Kommandos übernimmt.

Lader: ein Programm (evtl. Teil der Shell), das ein auf Datenträgern abgelegtes Programm lädt, also für die Ausführung vorbereitet. Dabei werden eventuell die endültigen Ablaufadressen festgelegt und die Aufteilung auf die Speichersegmente (Stack, Variablenbereich, Konstantenbereich) vorgenommen.

Library: s. Bibliothek

Linker: ein Programm, das C-Module und Bibliotheken zusammenbindet und als Einheit ablegt. Der Linker erzeugt relative oder absolute Adressen, die er in einer Symboltabelle den logischen Namen zuordnet. Auf dem Amiga sind alle Adressen relativ.

Lokal: intern

Lwert: Ein C-Ausdruck, der auf der linken Seite einer Zuweisung erscheinen darf.

Makro: eine Abkürzung für immer wiederkehrende Quellcodestrecken. Makros werden über Präprozessorbefehle definiert und vom Präprozessor expandiert. Makros können parametrisiert werden.

Modul: eine Sammlung von Deklarationen und Prozeduren, die als Einheit übersetzt wird. Allgemeiner: Prozedur

Optimierer: ein (nicht überall vorhandener) Ablaufschritt von Compilern. Der Optimierer versucht, den schon erzeugten Code zu verbessern, indem er z.B. Konstantenzuweisungen aus Schleifen entfernt oder unnötig häufige Speicherzugriffe vermeidet.

Parameter: synonym für Argument

Pointer: eine Maschinenadresse. C bietet im Gegensatz zu den meisten anderen Programmiersprachen Pointerarithmetik an.

Präprozessor: Teil eines C-Compilers oder eigenes Programm, das vor der Syntaxanalyse und vor der Codeerzeugung abläuft und ein Quellcodefilter darstellt. Der Präprozessor verfügt in C über eigene Befehle, die mit dem Zeichen # beginnen. Hauptziel des Präprozessors: Anpassung an Hardware- oder Compilereigenschaften.

Programm: in C eine Sammlung von Datendeklarationen, Präprozessorbefehlen und Prozeduren, eventuell in mehrere getrennt kompilierbare Module unterteilt.

Prozedur: Eine Programmstück, das das über einen Prozeduraufruf betreten und über ein (eventuell implizites) return wieder verlassen wird. Prozeduren können über Argumente parametrisiert werden. In C sind alle Prozeduren extern zueinander.

Register: Ein besonders schneller CPU-interner Speicher, der über spezielle Maschinenbefehle oder implizit (Akkumulator!) angesprochen wird.

Rwert: ein Ausdruck, der nur auf der rechten Seite einer Zuweisung erscheinen darf.

Routine: Synonym für Prozedur oder Funktion

Shell: Kommandoprozessor (in UNIX weitgehend programmierbar)

Seiteneffekt: der Effekt, den eine C-Funktion im globalen Bereich (Daten auf Platte, Veränderungen im globalen Speicher) hinterläßt und der nicht durch den Rückgabewert einer Funktion zu erklären ist. Seiteneffekte sollten vermieden werden, da sie schwer nachzuvollziehen sind.

Speicherklasse: die Speicherklasse einer Variablen legt fest, in welchem Speicher (Register, Stack, Datensegment) die Variable gehalten werden soll.

Standardroutinen: die eher logisch (zeichenweise, zeilenweise, formatorientiert) orientierten Ein-/Ausgaberoutinen der C-Bibliothek wie *fopen(), fprintf(), fgets(), putchar()* etc. Sie setzen auf den Basisroutinen auf. Sie verwenden Filepointer statt Dateinummern.

Stack: ein besonderer Speicherbereich, der zur Übergabe von Prozedurargumenten, zur Speicherung von Rückkehradressen und zur Haltung von internen Variablen verwendet wird.

Statisch: statische Variablen werden – selbst wenn sie intern sind – nicht auf dem Stack, sondern im Datensegment angelegt. Statische Namen sind nicht außerhalb des Blocks bzw. des Moduls bekannt.

String: eine Folge von *char*-Objekten, die durch eine binäre Null beendet wird. Stringkonstanten stehen zwischen doppelten Hochkommata.

Struktur: eine Folge von C-Objekten (Elementen) mit beliebigem Datentyp, die über einen gemeinsamen Namen ansprechbar sind.

Treiber: Ein hardwarenahes Programm, das periphere Geräte wie Laufwerke, Tastaturen, serielle Schnittstellen bedient.

Umlenkung (redirection): Einsatz des gleichen Programms für Ein-/Ausgabe auf Terminal oder in/aus Files. Kann mit C-Standardmitteln oder über den Kommandoprozessor angestoßen werden.

UNIX-IO: s. Basisroutinen

Unterprogramm: alle C-Prozeduren außer *main()* sind Unterprogramme. *main()* ist das Hauptprogramm.

Verbund: deutsche Bezeichnung für *union*. Sonderfall einer Struktur, bei der alle Elemente die gleiche Anfangsadresse haben.

Vorwärtsdeklaration: die Deklaration einer Funktion vor der ersten Verwendung ohne Angabe der formalen Argumente. Dient dazu, den Datentyp der Funktion festzulegen.

Zeichenkonstante: eine Konstante mit ganzzahligem Typ (*char, int, long*), die zwischen einfachen Hochkommata steht. In C gibt es für nichtdarstellbare Zeichen Ersatzdarstellungen, z.B. '*n*' oder '\012' für LF.

Zugriffsmaske: beim Anlegen eines Files wird eine Maske verwendet, die definiert, welche Zugriffs- und Besitzrechte die Benutzer eines Systems über diesen File haben. Die Prozeduren *creat(), open(), fopen(), umask(), chmod()* verwenden implizit oder explizit Zugriffsmasken.

Zusammengesetzter Datentyp: Arrays, Strukturen, Verbunde. Ihre Elemente bestehen aus Objekten mit einfachen, abgeleiteten oder selbst wieder zusammengesetzten Datentypen.

Stichwortverzeichnis

#include 15
#include-Befehle 158
#include-Files 158, 184

A
Ablaufphase 35
Additionsoperator + 66
Adreßkonstanten 135
Adreßoperator & 64
Adreßrechnung 117
Adreßvariablen 36
Animation 19
ANSI-Norm 26, 91, 126, 207
Argument 134, 141, 227
Argumenttypen 136
Argumentübergabe 138
Arithmetik 43, 207
Array-Parallelität 112
Arrayelement [] 61
Arrayindex 61
Arraynamen 38, 54
Arrays 35, 45, 105, 110, 227
ASCII-Bereich 155
ASCII-Codebereich 171
ASCII-Großbuchstaben 182
ASCII-Kleinbuchstaben 182
ASCII-Kontrollcodes 52
ASCII-Sortierung 215
ASCII-Strings 209
ASCII-Zeichen 52, 171
Assembler 26
Attribut 227
Audio-Kanäle 23
audio.device 23
Ausdrücke 59, 89, 227
Ausgabefunktionen 92
Austauschbarkeit 26
Aztec-Compiler 16

B
Basis-IO 164
Basisdatentyp 227
Basisprozeduren 92, 177
Basisroutinen 93, 227
Basistypen 47, 207
Batch-File 16, 17
Batchprozeduren 103
Bäume 127
BDOS-Call 106
Bedingungen 79
Befehl 227
Befehlsargumente 142
Befehlsschlüsselwort 75
Befehlstypen 75
Benutzerdialog 97
Benutzerschnittstelle 19
Betriebssystem 199
Betriebssystem-Schnittstellen 140
Bibliotheken 18, 228
Bibliotheksfunktionen 163, 184
Bibliotheksroutinen 156
Bildschirm 23
Binärdateien 178
Binder 106, 227
Bindungsrichtung 59
Bit 228

Blitter 19
Block 228
Blöcke 31, 89
Blockklammern 195
Boolean 51
bootlock.device 23
Byte 228

C
C-Befehle 34, 89
C-Bibliothek 91
C-Compiler 16
C-Objekte 55
C-Schlüsselwörter 55
call by address 138, 228
– by descriptor 228
– by reference 138, 150, 228
– by value 138, 150, 228
Cast-Operatoren 50, 62, 107
CLI 16f., 103
CLI-Icon 17
CLI-Schalter 17
CLI-Window 17
clipboard.device 23
clist.lib 19
Codegröße 204
Command Line Interpreter 16
Compilephase 35
Compiler 25
Compilersprachen 25, 26
Compilier-Fehler 17
Compilier-Phasen 16
console.device 23
console.lib 19
Copper 19
Copper-Liste 19
ctype.h 158

D
Datei-Verzeichnisse 20
Dateien 20, 228
Datenobjekt-Größe 63
Datenobjekte 42, 48
Datentyp-Hierarchien 45
Datentypen 18f., 42f., 227f.
Debugger 140, 228
Definition 228
Deklarationen 35, 76, 89, 123, 228
Deklarationsteil 190
Dekrementbefehle 203
Dekrementieren 228
Dekrementoperatoren 155
Device 24
Devices 15, 22f.
Devs-Directory 23
Dezimalsystem 51
DIN-Sortierung 215
Directory 228
diskfont.lib 19
Display-Elemente 20
Divisionsoperator / 67
do while-Schleife 83
dos.lib 19
Drucker 23

E
Editor 17, 25, 35
Eingabefunktionen 92
Ereigniseingaben 23
Erfassungsphase 35
exec.lib 19
Exor-Operator ^ 69
Extension 35
Extern 228

F
Fallmarke 81
Fehleranfälligkeit 27
Feldbeschreibungstabelle 216
FFP-Basic-Mathematik-Library 20
File 228
Filepointer 93
Flexibilität 26
Fließkommaarithmetik 49
Fließkommakonstanten 52
Fließkommazahlen 210, 229
Floppies 24
for-Schleife 83
Fragezeichenoperator ? 71
Funktionen 36, 45, 95, 133, 150, 229
Funktionsaufruf () 60
Funktionsaufrufoperator 135
Funktionsnamen 38, 54, 135
Funktionsprototyp 208
Funktionsvariablen 150

G
Gadgets 23
Game-Ports 23
gameport.device 23
Geschwindigkeit 25
Gleichheitsoperator == 69
Global 229
Grafikmodus 144

graphic.lib 19
Graphics-Library 21
Gruppierungsklammern 197
Gültigkeitsbereiche 56, 229

H
Harddisk 23
Hauptprogramm 229
Heapsegment 174
Hexadezimalkonstanten 51
Hilfsvariablen 192

I
icon.lib 19
Icons 19
IEEE-Darstellung 210
IEEE-Standard 20
Include-Dateien 18
Include-Files 15
info.lib 20
Initialisierung 121
Inkrementbefehle 203
Inkrementieren 229
Inkrementoperatoren 38, 155, 198
input.device 23
inputevent.device 23
Integer 229
Integerdivision 62
Intern 229
Interpretersprache 25
Interrupt 229
Intuition-Library 22
intuition.lib 20
IO-Fehler 174
IO-Funktionen 163
ISAM-Dateiverwaltung 173

J
janus.lib 20
jdisk.device 23

K
K&R 229
Kernighan & Ritchie 26
Kettenelement 129
keyboard.device 23
keymap.device 23
Kickstart-Diskette 23
Kommandointerpreter 141
Kommandoprozeduren 81
Kommandoprozessor 101, 229
Kommaoperator 72, 87
Kommentare 18, 30, 34
Kommentarteil 190
Kommunikationsanwendungen 200
Kommunikationskanäle 101
Kompilieren 159
Komplementoperator ~ 65
Konstanten 41f., 59, 139, 152
Konvertierungsregeln 48
Konvertierungsroutinen 158, 209
Konvertierungstabelle 215

L
Label 81, 88
Lader 229
Laufvariablen 192
layers.lib 20
Library 15, 19, 20f., 229
Link-Phasen 16
Linker 25, 35, 106, 227, 229
Linkphase 35
Listen 127
Logische Negation ! 65
Lokal 229
low-level-Routinen 92
Lwert 60, 230

M
Makrodefinition 153
Makrodeklarationen 158
Makros 133, 152, 204, 230
Maschinenadresse 105
Maschinenregister 58, 138
math.h 158
Mathematische Routinen 164
mathffp.lib 20
mathieeedoubbas.lib 20
mathtrans.lib 20
Maus 20
Menüs 20
Modul 230
MODULA 18
Modulbibliotheken 134
Modulo-Operator 67
Modulpakete 193
MSC-Compiler 19
Multiplikationsoperator * 66

N
Namensvergabe 192
narrator.device 23
Normierung 26
Notationen 52

O
Oder-Operator | | 68
Operationen 125
Operatoren 41, 59
Optimierer 230
Optimierungsphase 190
Overhead 204

P
parallel.device 23
Parallelport 23
Parameter 19, 134, 230
Pascal 18
Periperiegeräte 106
Pfeiloperator 62
pipes 101
Platzbedarf 25
Pointer 36, 45, 105, 129, 230
Pointer-Parallelität 112
Pointerarithmetik 106
Pointerkonstanten 38
Pointertechnik 203
Pointervariablen 107
Port 24
Postfix 35
Präprozessor 75, 151, 230
Präprozessoranweisungen 19
Präprozessorbefehle 34, 89
Präprozessorkommandos 161
Präprozessorteil 190
printer.device 23
Priorität 60
Programm 230
Programmierregeln 189
Programmierstil 189
Programmkopf 190
Programmkörper 190
Programmlogik 191
Programmoptimierung 203
Prozedur 230
Prozeduranwendungen 141
Prozedurargumente 38, 56
Prozeduraufrufe 79, 204
Prozedurdeklarationen 136, 159
Prozeduren 39, 133, 190
Prozeßkommunikation 101
prtbase.device 23
Punktoperator 61

Q
Quellbibliotheken 134
Quellcode 35, 151

R
Realismus 26
redirection 99
Register 230
Registervariablen 203f.
Rekursive Prozeduren 145
Restoperator % 67
Routine 230
Rückgabewerte 148
Rückkehradressen 38, 145
Rwert 60, 230

S
Scheduling 19
Schleifen 35, 83
Schleifenbedingung 83
Schleifenkonstruktionen 87
Schleifenkörper 83
Schleifenvariablen 84
Schlüsselwörter 89
Schnittstellen 22
Schreibregeln 51
Schrifttypen 19
Segmente 41
Seiteneffekte 155, 230
Semaphoren 101
serial.device 24
Seriellport 24
Shell 101, 230
Shiftoperator << 68
– >> 67
SideCar 16, 20
sizeof-Operator 63
Software-Uhr 20
Sortierbaum 146
Sound 23
Sourcecode 35
Speicherbereiche 41, 126
Speicherblöcke 184
Speicherklasse 230
Speicherklassen 56, 58
Speicherplatz 174
Speicherverwaltungsroutinen 164
Sprachausgabe 20, 23
Sprungbefehl 87
Sprungmarken 56, 58, 88
Stack 17, 138, 231
Stackadresse 139
Stackframe 145
Stacksegment 191
Standard-IO 164
Standardbibliothek 91

Standardroutinen 95, 231
Statisch 231
stdio.h 158
Stelligkeit 60
Sternoperator * 38, 64
String 231
String-Routinen 164
Stringkonstanten 210
Strings 32, 53
Stringverarbeitungsroutinen 164
structure tag 121
Strukturelement-Auswahl 61
Strukturen 45, 121, 130, 207, 231
Strukturmuster 121
Strukturnamen 54
Strukturregeln 190
Subtraktionsoperator - 66
Suffix 35
Synonyme 56
System-Kern 19
Systemfunktionen 164
Systemprogrammierung 199
Systemtabellen 200
Systemzeit 24

T
Tasks 19
Tastatur 19, 23
Terminal-IO 164
timer.device 24
timer.lib 20
trackdisk.device 22, 24
translator.lib 20
Treiber 106, 200, 231
typedef 56
Typerzwingungsoperator 50
Typprüfung 137

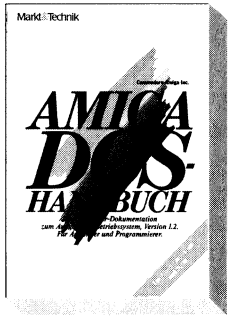
U
Übersichtlichkeit 26
Umlenkung 99, 231
Und-Operator && 68
Ungleichheitsoperator != 70
Universalität 27
UNIX 18, 26
Unterprogramme 133, 231
Unterroutinen 18

V
Variable 59
Variablen 18, 41, 42, 54
VariabVariablenadresse 64
Verbund-Namen 54
Verbunde 45, 121, 231
Vergleichsoperator 35
Vollständigkeit 26
Vorrangregeln 72
Vorwärtsdeklaration 137, 231
Vorzeichenoperator - 65

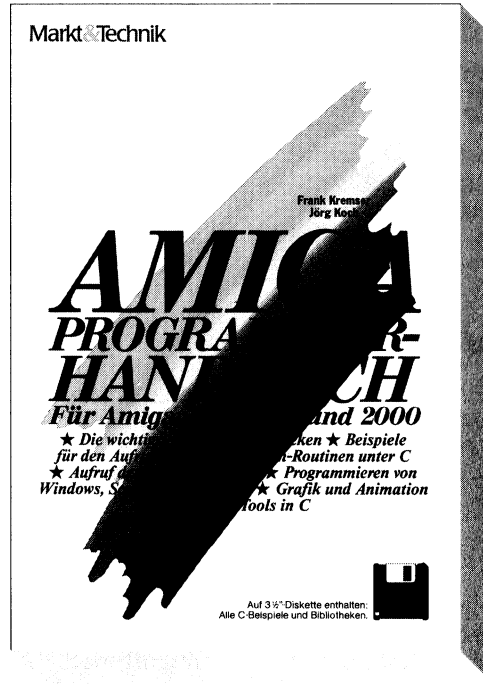
W
while-Schleife 83
Workbench 17

Z
Zeichenfilterung 200
Zeichenklassifizierung 164
Zeichenkonstanten 53, 231
Zeichenkonvertierung 164
Zeiger 105
Zeilennummern 30
Zugriffsmaske 231
Zuweisungen 78
Zuweisungsbefehle 78
Zuweisungsoperatoren 35, 70, 78

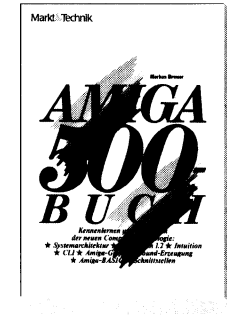
Bücher zum Amiga



Bantam Books
Das Amiga-DOS-Handbuch für Amiga 500, 1000 und 2000
1987, ca. 300 Seiten
Die Pflichtlektüre für jeden Commodore-Amiga-Anwender und Programmierer: eine Entwickler-Dokumentation zum Amiga-DOS-Betriebssystem, Version 1.2. Programmierung, interne Datenstruktur und Diskettenhandling. Mit diesem Buch lernen Sie das mächtige Amiga DOS schnell und sicher zu beherrschen. Alle Möglichkeiten des Systems, bis hin zum »Multi-Tasking« werden ausführlich und anschaulich beschrieben.
Best.-Nr. 90465
ISBN 3-89090-465-3
ca. DM 59,–



Kremser/Koch
Amiga Programmierhandbuch
1987, ca. 300 Seiten, inkl. Diskette
Eine tolle Einführung in die »Interna« des Amiga: Die wichtigsten Systembibliotheken, die das Betriebssystem zur Verfügung stellt, werden anhand vieler Beispiele erklärt. Aus dem Inhalt: Aufruf der Betriebssystem-Routinen unter C, Aufruf der DOS-Funktionen, Programmieren von Windows, Screens und Gadgets, Grafik und Animation, Tips und Tools in C.
Best.-Nr. 90491
ISBN 3-89090-491-2
DM 69,–



M. Breuer
Das Amiga-500-Handbuch
1987, ca. 450 Seiten
Eine ausführliche Einführung in die Bedienung des Amiga 500. Kennenlernen und Anwenden der neuen Computer-Technologie: Systemarchitektur, Workbench 1.2, Intuition, CLI, Amiga-Grafik, Sound-Erzeugung, Amiga-BASIC und Schnittstellen. Neben dem Handbuchteil mit vielen Bildschirmfotos und Übersichtstabellen, die Ihnen beim täglichen Einsatz helfen, schnell und reibungslos zu arbeiten, enthält das Buch eine ausführliche Beschreibung des Amiga 500 und seines Zubehörs.
Best.-Nr. 90522
ISBN 3-89090-522-6
DM 49,–

707328

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bitte schneiden Sie diesen Coupon aus, und schicken Sie ihn in einem Kuvert an:
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar

Computerliteratur und Software vom Spezialisten

Vom Einsteigerbuch für den Heim- oder Personalcomputer-Neuling über professionelle Programmierhandbücher bis hin zum Elektronikbuch bieten wir Ihnen interessante und topaktuelle Titel für

• Apple-Computer • Atari-Computer • Commodore 64/128/16/116/Plus 4 • Schneider-Computer • IBM-PC, XT und Kompatible
sowie zu den Fachbereichen Programmiersprachen • Betriebssysteme (CP/M, MS-DOS, Unix, Z80) • Textverarbeitung • Datenbanksysteme • Tabellenkalkulation • Integrierte Software • Mikroprozessoren • Schulungen.
Außerdem finden Sie professionelle Spitzen-Programme in unserem preiswerten Software-Angebot für Amiga, Atari ST, Commodore 128, 128D, 64, 16, für Schneider-Computer und für IBM-PCs und Kompatible!
Fordern Sie mit dem nebenstehenden Coupon unser neuestes Gesamtverzeichnis und unsere Programmservice-Übersichten an, mit hilfreichen Utilities, professionellen Anwendungen oder packenden Computerspielen!

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2,
8013 Haar bei München, Telefon (089) 4613-0

Adresse:

Name

Straße

Ort

Bitte schicken Sie mir:
☐ Ihr neuestes Gesamtverzeichnis
☐ Eine Übersicht Ihres Programmservice-Angebotes aus der Zeitschrift

☐ Außerdem interessiere ich mich für folgende/n Computer:

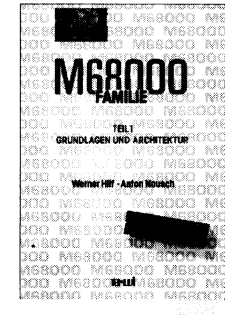
(PS: Wir speichern Ihre Daten und verpflichten uns zur Einhaltung des Bundesdatenschutzgesetzes)

Markt&Technik Verlag AG
– Unternehmensbereich Buchverlag –
Hans-Pinsel-Straße 2
D-8013 Haar bei München

Bücher zum Amiga

Prof. D. A. Lien
Amiga: Programmier-Praxis mit MS BASIC
1986, 400 Seiten
Bestseller! Eine systematische und lebendige Einführung in MS BASIC unter der komfortablen Mausbedienung und Fensteroberfläche des Amiga. Mit über 60 Musterprogrammen zu den Befehlen. Zeigt Amiga-typische Anwendungen: bewegte/farbige Grafiken; Musik- und Sprachausgabe, Strings, Felder, Mathematik, Dateibehandlung, Ein-/Ausgabe sowie »Entwurf von Programmen«.
Best.-Nr. 80369
ISBN 3-921803-69-1
DM 59,–

H.-R. Henning
Programmieren mit Amiga-BASIC
1987, ca. 360 Seiten, inkl. Diskette
Einführung in die Programmierung des Amiga-BASIC: Grafik, Sprites, Sprachausgabe, sequentielle Dateien, Fenstertechnik, Musik, Tips und Tricks.
Hard- und Software-Anforderungen: Amiga 500, 1000 oder 2000 mit 512 Kbyte Arbeitsspeicher, gegebenenfalls ein grafikfähiger Matrixdrucker und ein Joystick, Amiga-BASIC von Microsoft
Best.-Nr. 90434,
ISBN 3-89090-434-3
DM 59,–



W. Hilf/A. Nausch
M68000-Familie: Teil 1 Grundlagen und Architektur
1984, 568 Seiten
Ausbildungs- und Entwicklungstext mit allen notwendigen Informationen über den M68000.
Best.-Nr. 80316
ISBN 3-921803-16-0
DM 79,–

W. Hilf/A. Nausch
M68000-Familie: Teil 2 Anwendungen und 68000-Bausteine
1985, 400 Seiten
In vielen Programmierbeispielen liefert dieses Buch die Praxis der in Teil 1 vermittelten Theorie.
Best.-Nr. 80330
ISBN 3-921803-30-6
DM 69,–

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

707327

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Amiga: C in Beispielen

Die Autoren:

EDGAR HUCKERT wurde 1950 in Merzig/Saarland geboren. Studium der Sprachwissenschaft in Saarbrücken und Toulouse. Promotion 1979 über ein Thema aus dem Bereich Computer-Linguistik. Seit 1974 wissenschaftlicher Mitarbeiter in Sonderforschungsbereichen an den Universitäten Saarbrücken und Heidelberg. Seit 1979 Entwicklungsingenieur, vorwiegend in der graphischen Industrie, mit Schwerpunkt systemnahe Software. Aktiver Jazzmusiker (Saxophon).

FRANK KREMSER, geboren 1967, erwarb auf vielen Gebieten der Informatik Grundkenntnisse, die er in Form von Programmen und Berichten in Fachzeitschriften einem breiten Publikum zugänglich gemacht hat.

C hat sich in den letzten Jahren als die wichtigste universelle höhere Programmiersprache herausgestellt.
Das vorliegende Buch ist eine praxisgerechte Einführung in C, jedoch nicht für den absoluten Programmieranfänger geschrieben. Um auch Umsteigern aus anderen Programmiersprachen das Lesen und Arbeiten zu erleichtern, wurde das Modell amerikanischer Lehrbücher gewählt, in denen auf ein Kapitel »getting started« die systematischen Kapitel folgen. Wenn Sie also schnell zu einem laufenden C-Programm kommen wollen, sollten Sie sich das Anfangskapitel »Erste C-Programme« ansehen und dann das Buch eher im Sinne eines Handbuchs benutzen. Wenn Sie weniger Erfahrung mit C haben, arbeiten Sie das Buch am besten von vorne nach hinten durch.
Im einzelnen umfaßt das Buch folgende Abschnitte:

- Der Amiga und C
- Warum C?
- Erste C-Programme
- Daten und Datentypen
- Operatoren
- Befehle
- Eingabe und Ausgabe
- Arrays und Pointer
- Strukturen und Verbunde
- Prozeduren
- Der C-Präprozessor
- Die C-Bibliothek
- Programmierstil in C-Programmen
- Systemprogrammierung
- Programmoptimierung in C
- Die ANSI-Norm für C
- Kleines Glossar

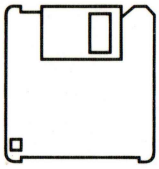
Abgerundet wird das Buch durch eine Sammlung von C-Programmen, wie zum Beispiel Konvertierroutinen, DIN-gerechte Sortierung oder ein Maskengenerator, die Sie zusammen mit den weiteren Listings im Buch auf der beigefügten Diskette finden.

Hardware-Voraussetzungen:
Amiga 500, 1000 oder 2000 mit 512 Kbyte RAM-Speicher

Software-Voraussetzungen:
C-Compiler wie Lattice C oder Aztec C

ISB N 3-89090-539-0

Markt&Technik



DM 69,-
sFr 63,50
öS 538,20